RIA 3/29/96

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	29 Feb 96	Final Report: 1 Jun 95 - 30 Nov 95

4. TITLE AND SUBTITLE

Artificial Intelligence and Operations Research
Timberline, Oregon Workshop: June 6 - 10, 1995

5. FUNDING NUMBERS

G: F49620-95-1-0373
PR-TA: C628/00

6. AUTHOR(S)

Matthew L. Ginsberg

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Oregon
Eugene, OR 97403

AFOSR-TR-96
0394

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFOSR/NM
110 Duncan Avenue Suite B115
Bolling AFB, DC 20332-0001

Program Manager:
Dr. Abraham Waksman

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The first international joint workshop on artificial intelligence and operations research was held successfully at Timberline Lodge in Timberline, Oregon on June 6 - 10, 1995. The workshop was supported by ARPA and AFOSR, and this report consists of proceedings of the conference.

19960726 088

14. SUBJECT TERMS

15. NUMBER OF PAGES
135

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCL	UNCL	UNCL	

Artificial Intelligence and Operations Research

Matthew L. Ginsberg

CIRL
1269 University of Oregon
Eugene, OR 97403-1269
ginsberg@cirl.uoregon.edu

Final technical report

Names and phone numbers of key personnel:

Technical:	Matthew L. Ginsberg	503-346-0471	ginsberg@cirl.uoregon.edu
Business:	Gary Chaffins	541-346-2395	

1 Summary

The first international joint workshop on artificial intelligence and operations research was held successfully at Timberline Lodge in Timberline, Oregon on June 6–10, 1995. The workshop was supported by ARPA and AFOSR, and this report summarizes the activity.

Requests for attendance at the workshop far outnumbered our ability to accommodate people, and workshop participants were limited to approximately forty individuals from academia, industry, and government. These participants included five government observers and the technical leaders of both fields; the organization of the workshop was designed to maximize the exchange of information between the two communities. As such, the workshop program (a copy follows this summary) included parallel overview tutorials or AI and OR, along with specific tutorials on topics that had been identified as of interest to the participants. These covered learning and Markov decision processes.

The program also included a variety of technical talks that were selected on the basis of both the quality of their results and their accessibility to a joint audience. Copies of all of the presented papers are also included in this report. Finally, industrial participants described specific application areas where joint AI/OR techniques could be expected to be applicable. These included emergency shutdown procedures for nuclear power plants, aircraft assembly operations, and molecular design problems.

The workshop was viewed as an overwhelming success by all participants, and at least one joint AI/OR effort has already arisen out of it (joint work by Nemhauser and Crawford funded by AFOSR). A variety of AI participants (Crawford, Ginsberg, Kautz, McAllester, Selman) have begun to participate in the annual OR conferences as well, and technical exchanges between the two communities continue.

Agenda
AI/OR Workshop
Timberline Lodge, Oregon
June 6 - 10, 1995

Tuesday 6/6

1500	Welcome (Hooker)
1515	Tutorials (basic AI (Selman), basic OR (Hall))
1645	Break
1700	Logic-based methods for engineering design (Hooker)
1830	Dinner banquet [keynote speech: Ginsberg]

Wednesday 6/7

0900	AI tutorial: learning (Minton)
1030	Break
1045	OR tutorial: MDP's (Bell)
1230	Lunch
1400	Solving problems with hard and soft constraints using a stochastic algorithm for max-sat (Kautz)
1435	An approach to the maximum satisfiability problem that combines heuristics with branch and cut (Borchers)
1510	On finding solutions for extended Horn formulas (Schlipf)
1545	Break
1600	The progressive party problem: Linear programming and constraint programming compared (B Smith)
1635	The TSP phase transition (Walsh)

Thursday 6/8

0900	Heterogeneous conjunctive constraints (Fox)
0945	Integer programming for job shop scheduling and a related problem (Boyd)
1020	Break
1035	A constraint satisfaction approach to makespan scheduling (S Smith)
1110	Planning and scheduling of nuclear power plant outages (Gomes)
1210	Lunch
1330	Combining the large-step optimization with tabu-search: Applications to the job-shop scheduling problem (Lourenco)
1405	An overview of learning in the Multi-tac system (Minton)

1440 Break

1455 Improving Dewar assembly process plans (D Smith)

1540 Incorporating efficient OR algorithms in constraint-based scheduling (Nuijten)

1615 Panel: opportunities for collaboration (Nemhauser)

Friday 6/9

0900 Free morning

1230 Lunch

1345 Chemical engineering process design, scheduling and operation (Realff)

1430 Representation and search issues involving MDPs (Dean)

1505 Break

1520 Panel: experimental results (Crawford)

1640 Panel: competing search methodologies (McAllester)

Saturday 6/10

0900 Robust encodings of OR problems for genetic algorithms (Bean)

0935 Interdependence of methods and representations in design of software for combinatorial optimization (Fourer)

1010 Break

1025 Panel 4: government interest (Khosla)

1130 Adjourn for 1:30 flights back to East Coast

Proceedings of the Conference

Jiang, Yuejun; Kautz, Henry; Selman, Bart. Solving Problems with Hard and Soft Constraints Using a Stochastic Algorithm for MAX-SAT

Borchers, Brian; Mitchell, John E. An Approach to the Maximum Satisfiability Problem that Combines Heuristics with Branch and Cut

Schlipf, John S.; Annexstein, Fred S.; Franco, John V.; Swaminathan, R.P. On Finding Solutions for Extended Horn Formulas

Smith, Barbara M.; Brailsford, Sally C.; Hubbard, Peter M.; Williams, H. Paul. The Progressive Party Problem: Linear Programming and Constraint Programming Compared

Gent, Ian P.; Walsh, Toby. The TSP Phase Transition

Boyd, E. Andrew. Integer Programming for Job Shop Scheduling and a Related Problem

Cheng, Cheng-Chung; Smith, Stephen F. A Constraint Satisfaction Approach to Makespan Scheduling

Lourenco, Helena Ramalhinho; Zwijnenburg, Michiel. Combining the Large-Step Optimization with Tabu-Search: Applications to the Job-Shop Scheduling Problem

Minton, Steve; Allen, John A.; Wolfe, Shawn; Philpot, Andrew. An Overview of Learning in the Multi-TAC System

Baptiste, Philippe; Le Pape, Claude; Nuijten, Wim. Incorporating Efficient Operations Research Algorithms in Constraint-Based Scheduling

Dean, Thomas. Position Paper for First International Joint Workshop on Artificial Intelligence and Operations Research

Bean, James C.; Hadj-Alouane, Atidel; Norman, Bryan. Robust Encodings of OR Problems for Genetic Algorithms

Coullard, Collette; Fourer, Robert. Interdependence of Methods and Representations in Design of Software for Combinatorial Optimization

Solving Problems with Hard and Soft Constraints Using a Stochastic Algorithm for MAX-SAT

Yuejun Jiang, Henry Kautz, and Bart Selman
AT&T Bell Laboratories
Direct correspondence to:
Henry Kautz
600 Mountain Ave., Room 2C-407
Murray Hill, NJ 07974
{kautz}@research.att.com

Abstract

Stochastic local search is an effective technique for solving certain classes of large, hard propositional satisfiability problems, including propositional encodings of problems such as circuit synthesis and graph coloring (Selman, Levesque, and Mitchell 1992; Selman, Kautz, and Cohen 1994). Many problems of interest to AI and operations research cannot be conveniently encoded as simple satisfiability, because they involve both hard and soft constraints – that is, any solution may have to violate some of the less important constraints. We show how both kinds of constraints can be handled by encoding problems as instances of weighted MAX-SAT (finding a model that maximizes the sum of the weights of the satisfied clauses that make up a problem instance). We generalize our local-search algorithm for satisfiability (GSAT) to handle weighted MAX-SAT, and present experimental results on encodings of the Steiner tree problem, which is a well-studied hard combinatorial search problem. On many problems this approach turns out to be competitive with the best current specialized Steiner tree algorithms developed in operations research. Our positive results demonstrate that it is practical to use domain-independent logical representations with a general search procedure to solve interesting classes of hard combinatorial search problems.

1 Introduction

Traditional satisfiability-testing algorithms are based on backtracking search (Davis and Putnam 1960). Surprisingly few search heuristics have proven to be generally

useful; increases in the size of problems that can be practically solved have come mainly from increases in machine speed and more efficient implementations (Trick and Johnson 1993). Selman, Levesque, and Mitchell (1992) introduced an alternative approach for satisfiability testing, based on stochastic local search. This algorithm, called GSAT, is only a partial decision procedure – it cannot be used to prove that a formula is unsatisfiable, but only find models of satisfiable ones – and does not work on problems where the structure of the local search space yields no information about the location of global optima (Ginsberg and McAllester 1994). However, GSAT is very useful in practice. For example, it is the only approach that can solve certain very large, computationally hard, formulas derived from circuit synthesis problems (Selman, Kautz, and Cohen 1994). It can also solve randomly generated Boolean formulas that are two orders of magnitude larger than the largest handled by any current backtracking algorithm (Selman and Kautz 1993a).

The success of stochastic local search in handling formulas that contain thousands of discrete variables has made it a viable approach for directly solving logical encodings of interesting problems in AI and operations research (OR), such as circuit diagnosis and planning (Selman and Kautz 1993b). Thus, at least on certain classes of problems, it provides a general model-finding technique that scales to realistically-sized instances, demonstrating that the use of a purely declarative, logical representation is not necessarily in conflict with the need for computational efficiency. One issue that arises in studying this approach to problem-solving is developing problem encodings where a solution corresponds to a satisfying model (Kautz and Selman 1992), instead of having a solution correspond to a refutation proof (Green 1969). But for some kinds of problems no useful encoding in terms of propositional satisfiability can be found – in particular, problems that contain both hard and soft constraints.

Each clause in a CNF (conjunctive normal form) formula can be viewed as a constraint on the values (true or false) assigned to each variable. For satisfiability, all clauses are equally important, and all clauses must evaluate to "true" in a satisfying model. Many problems, however, contain two classes of constraints: hard constraints that must be satisfied by any solution, and soft constraints, of different relative importance, that may or may not be satisfied. In the language of operations research, the hard constraints specify the set of feasible solutions, and the soft constraints specify a function to be optimized in choosing between the feasible solutions. When both kinds of constraints are represented by clauses, the formula constructed by conjoining all the clauses is likely to be unsatisfiable. In order to find a solution to the original problem using an ordinary satisfiability procedure, it is necessary to repeatedly try to exclude different subsets of the soft constraints from the problem representation, until a satisfiable formula is found. Performing such a search through the space of soft constraints, taking into account their relative importance, can be complex and costly in a practical sense, even when the theoretical complexity of the

entire process is the same as ordinary satisfiability.

A more natural representation for many problems involving hard and soft constraints is *weighted maximum satisfiability* (MAX-SAT). An instance of weighted MAX-SAT consists of a set of propositional clauses, each associated with a positive integer weight. If a clause is not satisfied in a truth assignment, then it adds the cost of the weight associated with the clause to the total cost associated with the truth assignment. A solution is a truth assignment that maximizes the sum of the weights of the satisfied clauses (or, equivalently, that minimizes the sum of the weights of the unsatisfied clauses). Note that if the sum of the weights of all clauses that correspond to the soft constraints in the encoding of some problem is l, and each hard constraint is represented by a clause of weight greater than l, then assignments that violate clauses of total weight l or less exactly correspond to feasible solutions to the original problem. The basic GSAT algorithm can be generalized, as we will show, to handle weighted MAX-SAT in an efficient manner. An important difference between simple SAT and weighted MAX-SAT problems is that for the latter, but not the former, near (approximate) solutions are generally of value.

The main experimental work described in this paper is on Boolean encodings of *network Steiner tree problems*. These problems have many applications in network design and routing, and have been intensively studied in operations research for several decades (Hwang *et al.* 1992). We worked on a well-known set of benchmark problems, and compared our performance with the best published results. One of our implicit goals in this work is to develop representations and algorithms that provide state-of-the-art performance, and advance research in both the AI and operations research communities (Ginsberg 1994).

Not all possible MAX-SAT encodings of an optimization problem are equally good. For practical applications, the final size of the encoding is crucial, and even a low-order polynomial blowup in size may be unacceptable. The number of clauses in a straightforward propositional encoding of a Steiner tree problem is quadratic in the (possibly very large) number of edges in the given graph. We therefore developed an alternative encoding, that is instead linear in the number of edges. This savings is not completely free, because the alternative representation only approximates the original problem instance – that is, theoretically it might not lead to an optimal solution. Nonetheless, the experimental results we have obtained using this encoding and our stochastic local search algorithm are competitive in terms of both solution quality and speed with the best specialized Steiner tree algorithms from the operations research literature.

The general approach used in our alternative representation of Steiner problems is to break the problem down into small, tractable subproblems, pre-compute a set of near-optimal solutions to each subproblem, and then use MAX-SAT to assemble a global solution by picking elements from the pre-computed sets. This general technique is applicable to other kinds of problems in AI and operations research.

In a sense this paper describes a line of research that has come full circle: much of the initial motivation for our earlier work on local search for satisfiability testing came from work by Adorf and Johnston (1990) and Minton *et al.* (1990) on using local search for scheduling problems that did involve both hard and soft constraints. Thus, we turned a method for optimization problems into one for decision problems, and now are returning to optimization problems. However, instead of creating different local search algorithms for each problem domain, we translate instances from different domains into weighted CNF, and use one general, highly optimized search algorithm. Thus we retain the use of purely propositional problem representations, and our finely-tuned randomized techniques for escaping from local minima during search.

2 A Stochastic Search Algorithm

The GSAT procedure mentioned in the introduction solves satisfiability problems by searching through the space of truth assignments for one that satisfies all clauses (Selman, Levesque, and Mitchell 1992). The search begins at a random complete truth assignment. The neighborhood of a point in the search space is defined as the set of assignments that differ from that point by the value assigned to a single variable. Each step in the search thus corresponds to "flipping" the truth-value assigned to a variable. The basic search heuristic is to move in the direction that maximizes the number of satisfied clauses. Similar local-search methods to satisfiability testing has also been investigated by Hanson and Jaumard (1990) and Gu (1992).

Thus GSAT can already be viewed as a special kind of MAX-SAT procedure, where all clauses are treated uniformly, and which is run until a completely satisfying model is found. We have experimented with many modifications to the search heuristic, and currently obtain the best performance with the following specific strategy for picking a variable to change. First, a clause in the problem instance that is unsatisfied by the current assignment is chosen at random – the variable to be flipped will come from this clause. Next, a coin is flipped. If it comes up heads (with a probability that is one of the parameters to the procedure), then a variable that appears in the clause is chosen at random. This kind of choice is called a "random walk". If the coin comes up tails instead, then the algorithm chooses a variable from the clause that, when flipped, will cause as few clauses as possible that are currently satisfied to become unsatisfied. This kind of choice is called a "greedy" move. Note that flipping a variable chosen in this manner will always make the chosen clause satisfied, and will tend to increase the overall number of satisfied clauses – but sometimes will in fact decrease the number of satisfied clauses. This refinement of GSAT was called "WSAT" (for "walksat") in Selman, Kautz, and Cohen (1994).

The *weighted* MAX-SAT version of Walksat, shown in Fig. 1, uses the sum of the weights of the affected clauses in computing the greedy moves. The parameter

```
procedure Walksat(WEIGHTED-CLAUSES, HARD-LIMIT, MAX-FLIPS,
        TARGET, MAX-TRIES, NOISE)
M := a random truth assignment over the variables that
    appear in WEIGHTED-CLAUSES;
HARD-UNSAT := clauses not satisfied by M with weight ≥ HARD-LIMIT;
SOFT-UNSAT := clauses not satisfied by M with weight < HARD-LIMIT;
BAD := sum of the weight of HARD-SAT and SOFT-UNSAT;
TOPLOOP: for I := 1 to MAX-TRIES do
    for J := 1 to MAX-FLIPS do
        if BAD ≤ TARGET then break from TOPLOOP; endif
        if HARD-UNSAT is not empty then
            C := a random member of HARD-UNSAT;
        else C := a random member of SOFT-UNSAT; endif
        Flip a coin that has probability NOISE of heads;
        if heads then
            P := a randomly chosen variable that appears in C;
        else
            for each proposition Q that appears in C do
                BREAKCOUNT[Q] := 0;
                for each clause C' that contains Q do
                    if C' is satisfied by M, but not
                        satisfied if Q is flipped then
                        BREAKCOUNT[Q] + = weight of C'
                    endif
                endfor
            endfor
            P := a randomly chosen variable Q that appears in C and whose
                BREAKCOUNT[Q] value is minimal;
        endif
        Flip the value assigned to P by M;
        Update HARD-UNSAT, SOFT-UNSAT, and BAD;
    endfor
endfor
print "Weight of unsatisfied clauses is", BAD;
print M;
end Walksat.
```

Figure 1: **The Walksat procedure for weighted MAX-SAT problems.**

HARD-LIMIT is set by the user to indicate that any clause with that weight or greater should be considered to be a hard constraint. The algorithm searches for MAX-FLIPS steps, or until the sum of the weights of the unsatisfied clauses is less than or equal to the TARGET weight. If the target is not reached, then a new initial assignment is chosen and the process repeats MAX-TRIES times. The parameter NOISE controls the amount of stochastic noise in the search, by adjust the ratio of random walk and greedy moves. The best performance on the problems in this paper was found when NOISE = 0.2.

Walksat is biased toward satisfying hard constraints before soft constraints. However, while working on the soft constraints, one or more hard constraints may again become unsatisfied. Thus, the search proceeds through a mixture of feasible and infeasible solutions. This is in sharp contrast with standard operations research methods, which generally work by stepping from feasible solution to feasible solution. Such methods are at least guaranteed (by definition) to find a local minimum in the space of feasible solutions. On the other hand, there is no such guarantee for our approach. It therefore becomes an empirical question as to whether local search on a weighted MAX-SAT encoding of problems with both hard and soft constraints would work even moderately well.

Our initial test problems were encodings of airline scheduling problems that had been studied by researchers in constraint logic programming (CLP) (Lever and Richards 1994). The results were encouraging; we found solutions approximately 10 to 100 times faster than the CLP approach. However, for the purposes of the paper, we wished to work on a larger test set, that had been studied more intensively over a longer period of time. We found such a set of benchmark problems in the operations research community, as we describe in the next section.

3 Steiner Tree Problems

Network Steiner tree problem have long been studied in operations research (Hwang *et al.* 1992), and many well-known, hard benchmark instances are available. The problems we used can be obtained by ftp from the OR Repository at Imperial College (mscmga.ms.ic.ac.uk). We ran our experiments on these problems so that our results could be readily compared against those of the best competing approaches. A network Steiner tree problem consists of an undirected graph, where each edge is assigned a positive integer *cost*, and a subset of its nodes, called the Steiner nodes. The goal is to find a subtree of the graph that spans the Steiner nodes, such that the sum of the costs of the edges of the tree is *minimal*. Fig. 2 shows an example of a Steiner problem. The top figure shows the graph, where the Steiner nodes are nodes 1, 2, 3, 6, and 7. The weights are given along the edges. The bottom figure shows a Steiner tree connecting those nodes. Note that the solution involves two non-Steiner nodes

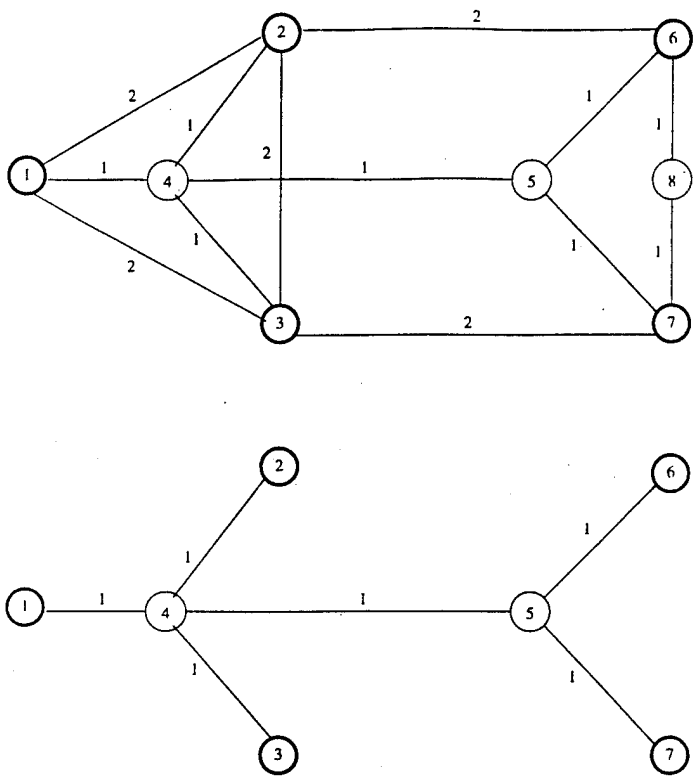


Figure 2: An example of a network Steiner problem and its solution.

(4 and 5). In general, finding such a Steiner tree is NP-complete.

There is a direct translation of Steiner problems into MAX-SAT. The encoding requires $2|E|^2$ variables, where $|E|$ is the number of edges in the entire graph. While this encoding is of theoretical interest, it is not practical for realistically-sized problems: even a quadratic blowup in the number of variables relative to the number of edges in original instance is simply too large. As we will see below, many of the problems we wish to handle contain over 10,000 edges, and we cannot hope to process a formula containing 100,000,000 variables! Therefore we developed an alternative encoding of Steiner tree problems that is only linearly dependent on the number of edges.

The intuition behind our encoding is that the original problem is broken down into a set of tractable subproblems; a range of near-optimal solutions to the subproblems are pre-computed; and then MAX-SAT is used to combine a selection of solutions to the subproblems to create a global solution. For Steiner tree problems, the subproblems are smaller Steiner trees that connect just *pairs* of nodes from the original Steiner set. Such two-node Steiner problems are tractable, because a solution is simply the shortest path between the nodes. A range of near-optimal solutions, *i.e.* the shortest path, the next shortest path, *etc.*, can be generated using a modified version of Dijkstra's

algorithm. This approach actually only approximates the original problem instance, because we do not generate *all* paths between pairs of nodes, but only the k shortest paths for some fixed k. (We discuss the choice of k below.) Pathological problem instances exist that require very non-optimal subproblem solutions. However, we shall see that the approach works quite well in practice.

We illustrate the encoding using the example from Fig. 2. First, we introduce a variable for each edge of the graph. For example, the edge between nodes 1 and 2 is represented by variable $e_{1,2}$. The interpretation of the variable is that if the variable is true, then the corresponding edge is part of the Steiner tree. To capture the cost of including this edge in the tree, we include a unit clause of the form $(\neg e_{1,2})$ with weight 2, the cost of the edge. This clause is soft constraint. Note that when this edge is included in the solution, *i.e.*, $e_{1,2}$ is true, this clause is unsatisfied, so the truth-assignment incurs a cost of 2. Similarly we have a clause for every edge.

Second, we list the Steiner nodes in an arbitrary order, and then for each successive pair of nodes in this list, we generate the k shortest paths between the nodes. We associate a variable with each path. For example, if $k = 2$, then the two shortest paths between Steiner nodes 1 and 2 are 1–2 and 1–4–2. We name the variables $p_{1,2}$ and $p_{1,4,2}$.

Third, we introduce hard constraints that assert that a solution must contain a path between each pair of Steiner nodes. For example, the clause $(p_{1,2} \vee p_{1,4,2})$ is a hard constraint, and therefore assigned a high weight (greater than the sum of all soft constraints). Hard constraints also assert that if a path appears in a solution, then the edges it contains appear. For example, for the path 1-4-2, we introduce the clauses $(p_{1,4,2} \supset e_{1,4})$ and $(p_{1,4,2} \supset e_{4,2})$. This concludes our encoding.

The encoding requires $|E|+k(|S|-1)$ variables, where $|E|$ is the number of edges in the graph, $|S|$ is the number of Steiner nodes, and k is the number of shortest paths pre-computed between each pair. The total number of clauses is $O(|E| + kL(|S|-1))$, where L is the maximum number of edges in any of the pre-computed paths.

4 Empirical Results

A good description of our benchmark problems appears in Beasley (1989). The set contains four classes (B, C, D, E) of problem instances of increasing size and complexity. We omitted class B because the problems are small and easy to solve. Each class has 20 instances.

Tables 1, 2, and 3 contain our results, as well as those of the two best specialized Steiner tree algorithms, as reported Beasley (1989) and Chopra *et al.* (1992). In the table, $|V|$ denotes the number of nodes in the graph, $|E|$ the number of edges, and $|S|$ the number of Steiner nodes. The columns labeled "Soln" give the weight of the best Steiner tree found by each method. The solutions found by Chopra *et al.* are globally

optimal, except for instance E18. For some problems we also give the second best solution (labeled "Soln2") found by Walksat, to indicate how effective the procedure can be in practice, since it may locate a near-global optimum in a very short time.

Walksat ran on a SGI Challenge with a 150 MHz MIPS R4400 processor. Beasley's algorithm ran on a Cray XMP, and Chopra's on a Vax 8700. A hyphen in the table in the case of Beasley's algorithm indicates that the problem was not solved after 21,600 seconds; in the case of Chopra's algorithm, it indicates that problem was not solved after 10 days.

We have not attempted to adjust the numbers for machine speed. Caution must be used in comparing different algorithms running on radically different kinds of hardware (the SGI has a RISC architecture, the Vax is CISC, and the Cray is a parallel vector processing machine). The SGI is rated is 136 MIPS, while the Vax is rated at 6 MIPS. This would indicate a ratio of 22 in relative speed; however, at least one user of both machines (Johnson 1994) reports a maximum speedup factor of 15 on combinatorial algorithms, with as small a factor as 3 on large instances. The Cray is rated 230 peak MIPS, which would appear to be faster than the SGI; however, Cray Research also reports that code that performs no vector processing at all runs at only 30 MIPS. Thus, differences in hardware could account for a speedup of between 3 and 22 when comparing Chopra's VAX to our SGI, and of between 0.6 and 4.5 when comparing Beasley's Cray to our SGI. In any case, this indicates that all of the differences in performance described below cannot be attributed entirely to differences in machine speed.

We found that we could obtain good solutions with a value of k, the number of pre-computed paths between pairs of nodes, of up to 150 for the smaller instances ($\leq$ 10 Steiner nodes), and up to 20 for the larger instances. The timing results for Walksat are averaged over 10 runs.

The running times in the table do not include the time to pre-compute the set of paths between successive Steiner nodes. This is reasonable because in practice one often deals with a fixed network, and wants to compute Steiner trees for many different subsets of nodes. For example, in teleconferencing applications, the network is fixed, and each problem instance involves finding a Steiner tree to connect a set of sites. Given a fixed network, one can pre-compute, using Dijkstra's algorithm, sets of paths between every pair of nodes.

From the tables we can see that for problems with up to 10 Steiner nodes, Walksat usually find an optimal solution at least as fast as the other two approaches, even allowing differences in machine speeds. For example, for D1 and D2, Walksat is about 100 times faster than the other two in reaching the global optimum. For D6, Walksat runs about 50 times faster than Beasley and 30 times faster than Chopra. The difference is particularly dramatic for E1, where Walksat finds the optimal solution in less than 1 second, and Beasley and Chopra both take over 1,000 seconds. On E2, Walksat takes about 800 seconds to reach the global optimum 214, which is

Problem Parameters				Beasley		Chopra *et al.*		Walksat			
ID	$\|V\|$	$\|E\|$	$\|S\|$	Soln	CPU secs (Cray XMP)	Soln	CPU secs (Vax 8700)	Soln1	CPU (SGI)	Soln2	CPU (SGI)
C1	500	625	5	85	113.57	85	27.3	85	1.11		
C2			10	144	5.84	144	811.7	144	72.69	146	30.57
C3			83	766	152.78	754	543.4	808	0.05		
C4			125	1094	3.61	1079	509.6	1128	0.09		
C5			250	1594	2.73	1579	473.9	1654	0.12		
C6		1000	5	55	48.55	55	48.9	55	3.41		
C7			10	106	4.44	102	83.2	102	3.02	103	2.95
C8			83	524	8.63	509	674.4	553	0.07		
C9			125	722	198.97	707	1866.3	754	0.09		
C10			250	1112	4.53	1093	245.6	1169	0.16		
C11		2500	5	34	188.02	32	333.3	32	0.44	34	0.22
C12			10	48	25.04	46	119.8	46	65.64	47	39.41
C13			83	265	166.53	258	9170.3	286	0.23		
C14			125	336	8.67	323	211.7	349	0.25		
C15			250	563	7.30	556	210.6	587	0.40		
C16		12500	5	11	32.37	11	10.1	11	6.25		
C17			10	20	24.17	18	98.0	18	19.50	19	6.52
C18			83	123	104.34	113	45847.7	130	4.89		
C19			125	155	86.48	146	116.9	165	5.25		
C20			250	269	157.80	267	14.9	278	5.79		

Table 1: Computation Results for Beasley's C class Steiner Tree Problems

comparable to Chopra's 6000 seconds (a ratio of 7.5). Walksat takes only about 28 seconds to reach a tree with weight 216, compared to Beasley who takes 7000 seconds to reach only 231. On E6, Walksat takes less than 2 seconds, compared to over 670 seconds for Chopra. A near-optimal solution takes less than 1 seconds, compared to 1700 seconds for Beasley.

Surprisingly, Walksat can locate some of the optimal and near-optimal solutions for the large E-class instances that cannot be found by Beasley in a reasonable amount of time. For example, for E12, Walksat finds a local optima of 68 which was not reached by Beasley within the time limit of 21,600 seconds. For E7, Walksat finds the global optimum of 145, while Beasley only reaches 157.

On problems with a larger numbers of Steiner nodes, Walksat usually produces less optimal solutions than the other two methods. The problem Walksat has on instances with a large number of Steiner nodes may due to the fact that the MAX-SAT encodings simply become too large to be processed efficiently. (For example,

Problem Parameters				Beasley		Chopra *et al.*		Walksat			
ID	$\|V\|$	$\|E\|$	$\|S\|$	Soln	CPU secs (Cray XMP)	Soln	CPU secs (Vax 8700)	Soln1	CPU (SGI)	Soln2	CPU (SGI)
D1	1000	1250	5	107	226.27	106	475.6	106	2.61	107	0.85
D2			10	228	252.47	220	283.5	220	1.54	227	0.98
D3			167	1599	21.85	1565	2290.1	1646	0.21		
D4			250	2170	11.71	1935	3529.0	2044	0.28		
D5			500	3360	11.76	3250	810.6	3419	0.53		
D6		2000	5	71	4065.69	67	2339.5	67	75.51	70	12.37
D7			10	103	18.71	103	99.7	103	0.47		
D8			167	1108	475.14	1072	6984.5	1180	0.35		
D9			250	1684	243.48	1448	4629.7	1585	0.41		
D10			500	2235	20.21	2110	1312.1	2219	0.72		
D11		5000	5	31	3290.48	29	1374.4	29	2.78	30	2.07
D12			10	42	48.04	42	305.0	42	0.79		
D13			167	520	36.06	500	1864.0	544	1.07		
D14			250	688	443.26	667	3538.4	740	0.74		
D15			500	1208	32.25	1116	1409.7	1193	1.70		
D16		25000	5	14	161.43	13	871.3	13	18.29		
D17			10	25	277.20	23	6965.2	23	735	24	20
D18			167	247	222.15	223	245192.1	262	20.48		
D19			250	384	256.15	310	878.3	359	21.52		
D20			500	544	1023.60	537	47.1	558	24.45		

Table 2: Computation Results for Beasley's D class Steiner Tree Problems

the number of flips per second goes down significantly on very large formulas.) Nonetheless, given the fact that Walksat is a completely general algorithm, as opposed to the specialized algorithms of Beasley and Chopra, it performs surprisingly well on these hard benchmark problems.

It is important to note that Walksat scales up to problems based on large graphs, especially when the set of Steiner nodes is relatively small. This should be contrasted with some other local-search style approaches to solving Steiner trees using simulated annealing (Dowsland 1991) and genetic algorithms (Kapsalis *et al.* 1993). Despite the fact that these local search algorithms were designed specifically for solving Steiner problems, they can only handle the smallest instances in the B and C classes. This has led Hwang *et al.* (page 172) to conclude that simulated annealing and hill-climbing (a form of local search) are ill-suited for Steiner tree problems. However, our work demonstrates that local search can in fact be successful for Steiner problems. Our positive results are due to both an effective problem encoding and the use of an

Problem Parameters				Beasley		Chopra *et al.*		Walksat			
ID	$\|V\|$	$\|E\|$	$\|S\|$	Soln	CPU secs (Cray XMP)	Soln	CPU secs (Vax 8700)	Soln1	CPU (SGI)	Soln2	CPU (SGI)
E1	1000	3250	5	115	1116.80	111	1149.6	111	0.54	113	0.35
E2			10	231	7124.10	214	6251.2	214	817.70	216	28.13
E3			417	4131	1346.05	4013	26468.4	4282	1.43		
E4			625	5208	378.66	5101	46007.6	5398	2.10		
E5			1250	8413	98.22	8128	12564.1	8518	3.95		
E6		5000	5	78	1760.49	73	678.0	73	1.71	78	0.81
E7			10	157	—	145	27124.0	145	5170.50	149	275.62
E8			417	2733	4459.30	2640	118617.5	2899	2.05		
E9			625	3721	18818,53	3604	24527.8	3913	2.65		
E10			1250	5899	311.57	5600	39260.7	5957	4.94		
E11		12500	5	39	3061.45	34	1900.6	34	622.71	35	7.47
E12			10	69	—	67	7199.7	68	5325.67	69	374.79
E13			417	1336	—	1280	207058.6	1417	7.21		
E14			625	1773	—	1732	29262.6	1884	8.69		
E15			1250	3008	457.98	2784	7666.0	3125	157.67		
E16		62500	5	15	7880.40	15	179.0	15	352.26	16	117.26
E17			10	26	445.69	25	36039.9	27	160.92		
E18			417	840	—	(563.03)	—	667	129.33		
E19			625	923	—	758	6371.8	853	132.70		
E20			1250	1376	14037.13	1342	272.2	1400	160.97		

Table 3: Computation Results for Beasley's E class Steiner Tree Problems

efficient implementation of our search procedure with a good stochastic technique for escaping from local minima.

5 Discussion and Conclusions

In this paper, we have shown how to adapt Walksat, a variant of the GSAT satisfiability testing algorithm, to handle *weighted* MAX-SAT problems. One of the problems in encoding optimization problems as propositional satisfiability problems is the difficulty of representing both hard and soft constraints. In a weighted MAX-SAT encoding, hard constraints simply receive a high weight (for example, larger than the sum of the soft constraints). Any solution where the sum of the weights of the violated clauses is less than that of any hard constraint is guaranteed to be feasible (*i.e.*, satisfies all hard constraints).

Another problem with translating optimization problems into satisfiability prob-

lems is handling numeric information. Even though in principle a polynomial transformation often exists, SAT encodings of realistic problem instances may become too large to solve. In our weighted MAX-SAT encoding, much of the numeric information in the problem instances can be captured effectively in the clause weights.

In order to test this approach, we considered a set of hard benchmark Steiner tree problems, and compared our results to specialized state-of-the-art algorithms. We chose the Steiner tree problem because of its long history and the public availability of a well-established set of benchmark instances. Our results showed that our weighted MAX-SAT strategy is competitive with specialized algorithms, especially on (possibly large and computationally difficult) instances involving small numbers of Steiner nodes. We must stress that we are not arguing that our approach is *the* best way to find Steiner trees. It is certainly the case that every particular class of combinatorial problems has some structure that can be best exploited by some specialized algorithm. The significance of our experiments is that they showed good performance using a completely general algorithm, that incorporates no heuristics specific to Steiner tree problems.

As mentioned above, the search performed by Walksat proceeds through truth-assignments that correspond to both feasible and infeasible solutions to the original optimization problem. This is an inherent aspect of our approach, simply because feasible solutions of the original problem may be several variable "flips" apart. Note that in constructing *specialized* local search algorithms for particular problem domains, one generally makes larger changes and only moves between feasible solutions. It is therefore surprising to discover how well Walksat performs. It is important to note that negative performance results would have argued against our overall approach of using a domain-independent logical representation with a general search procedure such as Walksat.

Part of the success of the approach is due to the particular MAX-SAT encoding we developed for the problems. In particular, our encoding is significantly shorter than a more direct one. The general approach we used, which is based on combining solutions from tractable subproblems, could also be useful for encoding other kinds of optimization problems. In particular, Crawford and Baker (1994) have observed that a direct SAT encoding of job-shop scheduling problems leads to formulas that are very large and hard to solve. It would be interesting to see if our piecewise encoding technique is applicable in the job-shop scheduling domain.

In conclusion, we have demonstrated that the use of efficient MAX-SAT encodings with a domain-independent stochastic local search algorithm is a promising approach for solving hard optimization problems in AI and operations research.

References

Adorf, H. M., and Johnston, M. D. (1990) A discrete stochastic neural network algorithm for constraint satisfaction problems. *Proceedings of the International Joint Conference on Neural Networks*, San Diego, CA.

Beasley, J. (1989) An SST-based algorithm for the Steiner Tree problems in graphs. *Networks* 19, 1-16.

Chopra, S., Gorres, E., and Rao, M. (1992) Solving the Steiner Tree problem on a graph using branch and cut. *ORSA Journal on Computing* 4(3), 3-18.

Crawford, J. M., and Baker, A.B. (1994). Experimental results on the application of satisfiability algorithms to scheduling problems. *Proceedings AAAI-94*, Seattle, WA, 1092-1097.

Davis, M., and Putnam, H. (1960). A computing procedure for quantification theory. *J. Assoc. Comput. Mach.* 7, 201–215.

Dowsland, K. (1991) Hill-climbing simulated annealing and the Steiner problem in graphs. *Eng. Opt.* 17, 91-107.

Ginsberg, M. and McAllester, D. (1994) GSAT and dynamic backtracking. *Proceedings KR-94*, Bonn, Germany, 226–237.

Ginsberg, M. (1994) Organizational meeting for AI/OR initiative, Oct. 1994.

Green, C. (1969) Application of theorem proving to problem solving. *Proceedings IJCAI-69*, Washington, DC, 219–239.

Gu, J. (1992) Efficient local search for very large-scale satisfiability problems. *Sigart Bulletin* 3(1), 8–12.

Hansen, P., and Jaumard, B. (1990) Algorithms for the maximum satisfiability problem. *Computing* 44, 279–303.

Hwang, F.K, Richards, D.S., and Winter, P. (1992) *The Steiner Tree Problem*, Amsterdam: North-Holland (Elsevier Science Publishers).

Johnson, D.S., (1994) Personal communication.

Kapsalis, A., Rayward-Smith, V., and Smith, G. (1993) Solving the graphical Steiner tree problem using genetic algorithms. *J. Oper. Res. Soc.* 44(4), 397-406.

Kautz, H., and Selman, B. (1992). Planning as satisfiability. *Proceedings ECAI-92*, Vienna, Austria.

Lever, J., and Richards, B. (1994) A CLP approach to flight scheduling problems. *Proceedings of the International Symposium on Methodologies for Intelligent Systems*, 1994.

Minton, S., Johnston, M.D., Philips, A.B., and Laird, P. (1990) Solving large-scale constraint satisfaction an scheduling problems using a heuristic repair method. *Proceedings AAAI-90*, Boston, MA, 17–24.

Papadimitriou, C.H., and Steiglitz, K. (1982) *Combinatorial Optimization.* Englewood Cliffs, NJ: Prentice-Hall.

Selman, B., Levesque, H.J., and Mitchell, D.G. (1992) A new method for solving hard satisfiability problems. *Proceedings AAAI-92,* San Jose, CA, 440–446.

Selman, B., and Kautz, H. (1993a) Domain-independent extensions to GSAT: solving large structured satisfiability problems. *Proceedings IJCAI-93*, Chambéry, France, 290–295.

Selman, B. and Kautz, H. (1993b) An empirical study of greedy local search for satisfiability testing. *Proceedings AAAI-93*, Washington, DC, 46–51.

Selman, B., Kautz, H., and Cohen, B. (1994) Noise strategies for local search. *Proceedings AAAI-94*, Seattle, WA, 1994.

Trick, M., and Johnson, D.S. (Eds.) (1993) Working notes of the DIMACS Algorithm Implementation Challenge, Rutgers University, New Brunswick, NJ.

An approach to the Maximum Satisfiability Problem that combines heuristics with Branch and Cut.

Brian Borchers[1] and John E. Mitchell[2]

Given a logical formula in conjunctive normal form (a conjunction of clauses, each of which is a disjunction of literals), the satisfiability problem is the problem of determining whether or not there is an assignment of the variables in the logical formula that makes the formula true. In the related maximum satisfiability problem, the objective is to find an assignment of the variables that maximizes the number of satisfied clauses.

Classical approaches to the satisfiability problem include the Davis–Putnam procedure (Davis and Putnam 1960), and resolution (Robinson 1965). More recently, several researchers (Blair et al. 1986, Harche et al. 1994, Hooker 1988a, Hooker 1988b, Hooker 1989) have formulated satisfiability problems as integer programming problems and then used branch and bound or branch and cut algorithms to solve the integer programming formulation of the satisfiability problem. Hooker (1988a) points out that the Davis–Putnam procedure is equivalent to the branch and bound approach, while resolution is equivalent to a simple cutting plane approach.

Variations on this theme include the column subtraction algorithm for solving the integer programming problem (Harche et al. 1994), and the use of horn clause resolution as a lower bounding technique within the branch and bound procedure (Gallo and Urbani 1989). However, all of these methods are aimed specifically at the satisfiability problem rather than the maximum satisfiability problem.

Other researchers (Hansen and Jaumard 1990, Selman et al. 1992, Resende and Feo 1994) discuss heuristics that are very effective in finding solutions to "yes" instances of the satisfiability problem and in finding good solutions to the maximum satisfiability problem. However, these algorithms are unable to prove the unsatisfiability of "no" instances of the satisfiability problem and they are unable to prove the optimality of solutions to the maximum satisfiability problem.

We have developed an approach to the maximum satisfiability problem that combines heuristics with integer programming techniques to obtain provably optimal solutions to maximum satisfiability problems. In this approach, a heuristic is first used to obtain a good solution to the maximum satisfiability problem. We then formulate the maximum satisfiability problem as an integer programming problem and use a branch and cut approach to solve the integer programming

[1]New Mexico Tech, Department of Mathematics, Socorro, NM 87801, borchers@nmt.edu
[2]Rensselaer Polytechnic Institute, Department of Mathematical Sciences, Troy NY 12180,. mitchj@rpi.edu

problem. The heuristic solution provides an incumbent solution that improves the performance of the branch and cut algorithm.

The main difference between our approach and branch and cut approaches to the satisfiability problem is in the formulation of the satisfiability problem as an integer programming problem. The satisfiability problem can be formulated as an integer feasibility problem:

$$\begin{array}{rcll} \sum_{y_i \in C_j^+} x_i + \sum_{y_i \in C_j^-} (1 - x_i) & \geq & 1 & j = 1 \ldots k \\ x_i & \in & \{0,1\} & i = 1..n \end{array} \tag{1}$$

Or, we can introduce an auxiliary variable w and obtain the integer programming problem:

$$\begin{array}{lrcll} \min & w & & & \\ \text{subject to} \quad \sum_{y_i \in C_j^+} x_i + \sum_{y_i \in C_j^-} (1 - x_i) + w & \geq & 1 & j = 1 \ldots k \\ x_i & \in & \{0,1\} & i = 1..n \\ w & \geq & 0 & \end{array} \tag{2}$$

However, in order to solve the the maximum satisfiability problem, we need to introduce an auxiliary variable for each clause:

$$\begin{array}{lrcll} \min & \sum_{j=1}^{k} w_j & & & \\ \text{subject to} \quad \sum_{y_i \in C_j^+} x_i + \sum_{y_i \in C_j^-} (1 - x_i) + w_j & \geq & 1 & j = 1 \ldots k \\ x_i & \in & \{0,1\} & i = 1..n \\ w_j & \geq & 0 & j = 1 \ldots k \end{array} \tag{3}$$

This formulation of the maximum satisfiability problem was examined by Cheriyan et al. (1994) in the special case of the two satisfiability problem in which each clause is restricted to two literals. For this special case, they developed a cutting plane algorithm that was able to solve large instances of the two satisfiability problem without resorting to branch and bound.

It is well known that the LP relaxation of (1) or (2) is a very weak formulation of the problem. By fixing the variables in clauses with only one variable at zero or one as needed to satisfy that clause and by setting each of the remaining x_i to 1/2, we can easily obtain a feasible solution. Harche et al. (1994) note that simple branch and bound on these integer programming formulations is exactly equivalent to the Davis–Putnam procedure with unit clause tracking, so that solving the LP relaxations serves no purpose except to find integer solutions.

In (3), the LP relaxation is not so useless. For example, consider the clauses:

$$\begin{array}{c} \bar{x_1} \\ x_1 \\ \bar{x_1} \\ x_1 \vee x_2 \\ x_1 \vee \bar{x_2} \end{array}$$

The LP relaxation of formulation (1) is infeasible. The LP relaxation of formulation (2) is feasible with a minimum value of 1/2, indicating that the clauses are not satisfiable. The LP relaxation of (3) has an optimal value of 3/2, indicating that at least two clauses must remain unsatisfied in any assignment of the variables.

We have implemented a preliminary version of our algorithm which uses the GSAT procedure (Selman et al. 1992) to obtain an incumbent solution to the Max–Sat problem, formulates the Max–Sat problem as an integer programming formulation using (3), tightens the formulation with resolution cuts and then uses the CPLEX branch and bound routine to solve the integer programming problem. We have used this code to solve a number of "no" instances of the satisfiability problem, including hard random 3-cnf problems with as many as 140 variables and 630 clauses and problems from the DIMACS collection of benchmarks with as many as 435 variables and 1037 clauses. In our experience, formulation (3) is not significantly harder to solve than formulation (1) or formulation (2). Furthermore, the good incumbent solutions provided by the GSAT heuristic are often helpful in speeding up the branch and bound process.

We are currently in the process of reimplementing this approach using the MINTO mixed integer programming optimizer. The new code will implement a true branch and cut procedure. We intend to perform extensive computational testing on this new implementation of our approach.

References

[1] C. E. Blair, R. G. Jeroslow, and J. K. Lowe. Some results and experiments in programming techniques for propositional logic. *Computers and Operations Research*, 13(5):633–645, 1986.

[2] J. Cheriyan, W. H. Cunningham, L. Tunçel, and Y. Wang. A linear programming and rounding approach to max 2-sat. Technical report, Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Canada N2L 3G1, 1994.

[3] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. Assoc. Comput. Mach.*, 7:201–215, 1960.

[4] G. Gallo and G. Urbani. Algorithms for testing the satisfiability of propositional formulae. *J. Logic Programming*, 7:45–61, 1989.

[5] P. Hansen and B. Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44:279–303, 1990.

[6] F. Harche, J. N. Hooker, and G. L. Thompson. A computational study of satisfiability algorithms for propositional logic. *ORSA Journal on Computing*, 6:423–435, 1994.

[7] J. N. Hooker. A quantitive approach to logical inference. *Decision Support Systems*, 4:45–69, 1988.

[8] J. N. Hooker. Resolution vs. cutting plane solution of inference problems: some computational experience. *Operations Research Letters*, 7(1):1–7, 1988.

[9] J. N. Hooker. Input proofs and rank one cutting planes. *ORSA Journal on Computing*, 1(3):137–145, 1989.

[10] M. G. C. Reseande and T. A. Feo. A GRASP for Satisfiability. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1994.

[11] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92), San Jose, CA*, pages 440–446, July 1992.

On Finding Solutions for Extended Horn Formulas

John S. Schlipf*, Fred S. Annexstein†, John V. Franco*, R. P. Swaminathan*†

Department of Computer Science
University of Cincinnati
Cincinnati, OH 45221–0008, USA.

Abstract

In this note we present a simple quadratic-time algorithm for solving the satisfiability problem for a special class of boolean formulas. This class properly contains the class of extended Horn formulas [1] and balanced formulas [2, 4]. Previous algorithms for these classes require testing membership in the classes; however, the problem of recognizing whether a formula is balanced is complex, and the

problem of recognizing whether a formula is extended Horn is not known to be polynomial time. Our algorithm requires no such test for membership.

Keywords: Algorithms, satisfiability, extended Horn formulas , balanced matrices, unit resolution

1 Introduction

Chandru and Hooker [1] introduced the class of extended Horn formulas and showed that unit resolution alone can determine whether or not a given extended Horn formula $\mathcal{I}$ has a satisfying truth assignment. By repeated variable assignments and unit resolutions one can obtain a satisfying truth assignment for $\mathcal{I}$. The results of [1] cannot, however, be applied unless it is known that $\mathcal{I}$ is extended Horn. Unfortunately, the problem of recognizing extended

Horn formulas is not known to be solved in polynomial time.

In this paper we extend the observations of Chandru and Hooker to show that extended Horn formulas can be solved in quadratic time without first having to recognize them. Our observation is based on the following two facts. The first follows from two results in [1]; the authors may have been aware of it, but they did not explicitly state it. The second fact follows immediately from the soundness of resolution.

Below, let T denote any backtrack search tree for an arbitrary formula $\mathcal{I}$ where each node represents a set of clauses closed under unit clause resolution.

*Research partially supported by ONR grant N00014-94-1-0382.
†Research partially supported by NSF grant CCR-93-09470.

Fact 1 *If $\mathcal{I}$ is extended Horn, then there is a path from every internal node of T to a terminal node representing a solution.*

Fact 2 *If the root of T is an unsatisfiable terminal node (i.e., the empty clause is an element of the root), then $\mathcal{I}$ is unsatisfiable even if $\mathcal{I}$ is not an extended Horn formula.*

Based on these facts we develop a polynomial-time Davis-Putnam-like algorithm that solves extended Horn and other classes of formulas. The algorithm has two salient features. First, it investigates only one path of tree T, starting at the root, using one level unit-resolution look-ahead to select direction through T. Second, it gives up if the look-aheads in both directions yield contradictions. On extended Horn formulas, our algorithm will always find a satisfying truth assignment, if one exists, or verify unsatisfiability if no satisfying assignment is possible. On other formulas the algorithm will either provide a satisfying truth assignment, verify unsatisfiability, or give-up. The algorithm also never gives up on hidden Horn formulas, simple extended Horn formulas [3], hidden extended Horn formulas, and balanced formulas [2].

We start with terminology and a review of extended Horn and balanced formulas. That is followed by our results and a section stating possible future directions.

2 Extended Horn and balanced formulas

In this paper all formulas are in Conjunctive Normal Form (CNF). Let $V = \{v_1, v_2, ..., v_n\}$ be a set of n variables. A literal is either a variable or its negation and is said to be a positive literal if it is unnegated or a negative literal if negated. A clause is a disjunction of literals and is represented here as a set of literals. If a clause has exactly one literal, it is called a *unit clause*. A formula is a conjunction of clauses and is represented as a set. A truth assignment is represented as a set of variables t with the interpretation that exactly those variables in t have value *true* and the rest have value *false*. A formula is satisfiable iff there is a truth assignment that causes all clauses of $\mathcal{I}$ to have truth value *true*.

The following is a well known result that is the basis of the Davis-Putnam Procedure.

Fact 3 *Let $\mathcal{I}$ be a CNF formula and let v be a variable associated with either a positive or negative literal in $\mathcal{I}$. Let $\mathcal{I}_1 = \{C - \{v\} : C \in \mathcal{I}, \bar{v} \notin C\}$, and let $\mathcal{I}_2 = \{C - \{\bar{v}\} : C \in I, v \notin C\}$. Then $\mathcal{I}$ is satisfiable iff $\mathcal{I}_1$ or $\mathcal{I}_2$ is satisfiable.* □

Recursive application of Fact 3 results in a usual backtrack search tree where each node represents a split into at most two subproblems. A path in the tree represents a partial assignment

of truth values to variables. The formula associated with a node p in the tree is denoted by $\mathcal{I}(p)$. A node p of the tree is terminal if and only if either (i) the empty clause is an element of $\mathcal{I}(p)$, in which case $\mathcal{I}$ is not satisfied by any truth assignment containing the partial assignment represented by the path to that node from the root, or (ii) $\mathcal{I}(p)$ is the empty set, in which case $\mathcal{I}$ is satisfied by any truth assignment containing the partial assignment represented by the path to that node from the root. There are many possible backtrack trees for $\mathcal{I}$.

When one resolves on a unit clause, then $\mathcal{I}_1$ or $\mathcal{I}_2$ or both contain the empty clause. In our backtrack trees (called T above) we collapse into single nodes all nodes based on successive selections of variables from unit clauses.

The class of extended Horn formulas was introduced by Chandru and Hooker, motivated by results from linear programming. The characterization below is taken from [3].

Definition 2.1 Let C be a clause built from a variable set V and let R be a rooted directed tree with root s (i.e., a directed tree with all edges directed awa y from s) and with edges (uniquely) labeled with variables in V. Then C is *extended Horn w.r.t. R* if the positive literals of C label a dipath P of R and the set of negative literals in C label an edge-disjoint union of dipaths $Q_1, Q_2, ..., Q_t$ of R with exactly one of the following conditions satisfied:

1. $Q_1, Q_2, ..., Q_t$ start at the root s.
2. $Q_1, Q_2, ..., Q_{t-1}$, (say), start at the root s, and Q_t and P start at a vertex $q \neq s$.

Clause C is *simple extended Horn w.r.t. R* if it is extended Horn w.r.t. R and Condition 1 above is satisfied. A CNF formula $\mathcal{I}$ is (*simple*) *extended Horn w.r.t. R* if each clause $C \in \mathcal{I}$ is (simple) extended Horn w.r.t. R. A formula is (*simple*) *extended Horn* if it is (simple) extended Horn w.r.t. some such rooted directed tree R.

Fact 1 follows from two results of Chandru and Hooker:

Lemma 2.1 [1] *Suppose $\mathcal{I}$ is an extended Horn formula containing no unit or empty clauses. Then $\mathcal{I}$ is satisfiable.*

Proof Sketch: Assign truth value *true* to variables labeling alternating levels of the extended Horn tree, starting with the second. □

Lemma 2.2 [1] *If $\mathcal{I}$ is an extended Horn formula and v is any variable, then $\mathcal{I}_1 = \{C - \{v\} : C \in \mathcal{I}, \bar{v} \notin C\}$, and $\mathcal{I}_2 = \{C - \{\bar{v}\} : C \in I, v \notin C\}$ are also extended Horn.*

Proof Sketch: Eliminating the variable v corresponds to contracting the edge labeled v in the extended Horn tree; the extended Horn property is preserved. □

Remark. The definitions of "extended Horn" [1] and "simple extended Horn" [3] required that the set of negative literals of a clause C be partitioned into disjoint sets $Q_1, Q_2, \ldots, Q_t$, forcing the corresponding paths to be edge disjoint. It is easy to check that Lemmas 2.1 and 2.2, and thus also Fact 1, hold even if the Q_is are not disjoint. Thus, their techniques, as well as ours, apply to a class of formulas larger than the original extended Horn class.

Definition 2.2 Let $\mathcal{I}$ denote a CNF formula.

1. In $\mathcal{I}$, *reversing the polarity* of variable v is the process of replacing every occurrence of v in $\mathcal{I}$ with $\overline{v}$ and replacing every occurrence of $\overline{v}$

   with v.

2. The formula $\mathcal{I}$ is *hidden extended Horn* if reversing the polarities of some of its variables will yield an extended Horn formula.

Another class of formulas for which linear programming motivated a polynomial time satisfiability test is the class of *balanced* formulas defined by [2, 4].

Definition 2.3 Consider a CNF formula $\mathcal{I} = \{C_1, \ldots, C_m\}$ on a variable set $V = \{v_1, \ldots, v_n\}$. Associate with $\mathcal{I}$ a $m \times n$ $(0, \pm 1)$-matrix M as follows: The rows of M are indexed on $\mathcal{I}$ and the columns are indexed on V such that the entry M_{ij} is a $+1$ if $v_j \in C_i$, a -1 if $\overline{v}_j \in C_i$ and a 0 otherwise. Then $\mathcal{I}$ is a *balanced formula* if in every submatrix of M with exactly two nonzero entries per row and per column, the sum of the entries is a multiple of four.

3 A single look-ahead algorithm

In this section we present an algorithm for solving a class of propositional satisfiability problems properly containing some well known classes such as extended Horn, hidden extended Horn, hidden Horn, simple extended Horn, and balanced formulas. This algorithm does not require the recognition of any members of this class. The algorithm **SLUR**, which stands for Single Look-ahead Unit Resolution, is stated after the following algorithm **UCR** which applies Unit Clause Resolution as many times as possible to a given CNF formula.

Algorithm **UCR** $(\mathcal{I})$
Input: A CNF formula $\mathcal{I}$
Output: A CNF formula $\mathcal{I}'$ without unit clauses

While there is a unit clause $\{l\}$ in $\mathcal{I}$ do the following:

If l is a positive literal set $\mathcal{I} := \{C - \{\bar{l}\} : C \in \mathcal{I}, l \notin C\}$.

Otherwise set $\mathcal{I} := \{C - \{l\} : C \in \mathcal{I}, \bar{l} \notin C\}$.

Output $\mathcal{I}$.

End Algorithm **UCR**

Algorithm **SLUR**($\mathcal{I}$)
Input: A CNF formula $\mathcal{I}$
Output: A satisfying truth assignment for the variables in $\mathcal{I}$, "unsatisfiable," or "give up"

Initialize $t := \emptyset$.
Initialize $\mathcal{I}$:=**UCR**($\mathcal{I}$).
If $\emptyset \in \mathcal{I}$ then output "unsatisfiable" and halt.
While $\mathcal{I}$ is not empty do the following:
Select a variable v appearing as a literal of $\mathcal{I}$.
Set $\mathcal{I}_1$:=**UCR**($\{C - \{v\} : C \in \mathcal{I}, \bar{v} \notin C\}$).
Set $\mathcal{I}_2$:=**UCR**($\{C - \{\bar{v}\} : C \in \mathcal{I}, v \notin C\}$).
If $\emptyset \in \mathcal{I}_1$ and $\emptyset \in \mathcal{I}_2$ then output "give up" and halt.
Otherwise, if $\emptyset \notin \mathcal{I}_1$ set $\mathcal{I} := \mathcal{I}_1$.
Otherwise, set $t := t \cup \{v\}$, and set $\mathcal{I} := \mathcal{I}_2$.
Output $t \cup \{v : v$ was eliminated by **UCR** along the chosen path$\}$.
End Algorithm **SLUR**

Clearly, the number of search tree nodes visited by **SLUR** is linear in the size of $\mathcal{I}$. Hence, since **UCR** can be performed in linear time, **SLUR** is a quadratic time algorithm. The following propositions assert the correctness of **SLUR**.

Proposition 3.1 *If algorithm* **SLUR**($\mathcal{I}$) *returns a truth assignment t, then t satisfies* $\mathcal{I}$. *If algorithm* **SLUR**($\mathcal{I}$) *returns "unsatisfiable," then* $\mathcal{I}$ *is not satisfiable.*

Proof: The first statement follows from Fact 3 and the fact that a truth assignment is returned for a node p such that $\mathcal{I}_p = \emptyset$. The second statement follows from Fact 2. □

We say that algorithm **SLUR** *solves* formula $\mathcal{I}$ if it never gives up on input $\mathcal{I}$ for any order of variable selection.

Proposition 3.2 *Algorithm* **SLUR** *solves* $\mathcal{I}$ *if* $\mathcal{I}$ *is (i) extended Horn, (ii) hidden extended Horn, (iii) simple extended Horn, or (iv) balanced.*

Proof: Part (i) follows from Fact 1. Part (ii) follows from (i) and the fact that reversing the polarity of literals corresponding

to any variable does not have any significant impact on the operation of **SLUR**. Part (iii) follows from the fact that simple extended Horn formulas are a subclass of extended Horn formulas. Part (iv) follows from the fact that **SLUR** is an elaboration of the algorithm used to construct satisfying assignments for balanced formulas in [2]. □

The examples below show that **SLUR** solves a larger class of formulas than hidden extended Horn (even as generalized in the Remark following Lemma 2.2), so **SLUR** cannot be used to recognize hidden extended Horn formulas. The proofs of both propositions are easy, and therefore omitted.

Proposition 3.3 *Suppose that algorithm* **SLUR** *solves formula $\mathcal{I}$, and let $\mathcal{I}'$ be a set of clauses that are logical consequences of $\mathcal{I}$. Then algorithm* **SLUR** *also solves formula $\mathcal{J} = \mathcal{I} \cup \mathcal{I}'$.*

Proposition 3.4 *Suppose $\mathcal{I}$ is a formula containing clauses C_1, C_2, C_3, C_4 and a, b are variables where $\{a, b\} \subseteq C_1$, $\{a, \overline{b}\} \subseteq C_2$, $\{\overline{a}, b\} \subseteq C_3$, and $\{\overline{a}, \overline{b}\} \subseteq C_4$. Then $\mathcal{I}$ is not hidden extended Horn (even as generalized in the Remark following Lemma 2.2).*

Example 1 Let $\mathcal{I}$ be any non-empty formula solved by **SLUR**, let C be any clause in $\mathcal{I}$, let a, b be two variable symbols, and let

$$\mathcal{J} = \mathcal{I} \cup \{C \cup \{a, b\}, C \cup \{a, \overline{b}\}, C \cup \{\overline{a}, b\}, C \cup \{\overline{a}, \overline{b}\}\}.$$

Then **SLUR** solves $\mathcal{J}$ (by Proposition 3.3) but $\mathcal{I}'$ is not hidden extended Horn (by Proposition 3.4).

Example 2 Let V consist of three or more variable symbols and let $\mathcal{I}$ be the CNF formula asserting that exactly an even number of variables of V are true. E.g., for $V = \{a, b, c\}$ we have

$$\mathcal{I} = \{\{\overline{a}, b, c\}, \{a, \overline{b}, c\}, \{a, b, \overline{c}\}, \{\overline{a}, \overline{b}, \overline{c}\}\}.$$

Then Fact 1 holds for $\mathcal{I}$ trivially since any assignment

of truth values to all but one of the variables in V can be extended to an assignment of truth values making $\mathcal{I}$ true — just determine the truth value of the remaining variable by parity. Hence **SLUR** solves $\mathcal{I}$. But $\mathcal{I}$ is not hidden extended Horn (by Proposition 3.4). Nor is $\mathcal{I}$ balanced (by inspection of the matrix). Moreover, $\mathcal{I}$ is not equivalent to any proper subset of itself, so $\mathcal{I}$ cannot be constructed from a hidden extended Horn or balanced formula as in Example 1.

4 Conclusion

Researchers have used techniques from linear programming to identify some classes of boolean formulas for which satisfiability can be determined quickly. Such classes include extended Horn and balanced formulas. Indeed, for these classes, unit clause resolution is sufficient for testing satisfiability. However, the applicability of these techniques is limited by the complexity of identifying such formulas. In fact, the recognition problems for these formula classes turn out to be interesting combinatorial challenges. We have shown in this note that, in regards to the problem of finding satisfying assignments, the recognition problems are moot. Moreover, this work begins a study of the much broader class of formulas satisfying Fact 1.

References

[1] V. Chandru and J. N. Hooker. 1991. Extended Horn sets in propositional logic. *J. ACM* **38**, pp. 205–221.

[2] M. Conforti and G. Cornuéjols. 1992. A class of logical inference problems solvable by linear programming. *FOCS* **33**, pp. 670–675.

[3] R.P. Swaminathan and D.K. Wagner. 1991. The arborescence–realization problem. *Discrete Applied Mathematics*, to appear.

[4] K. Truemper, 1978. On balanced matrices and Tutte's characterization of regular matroids, preprint.

The Progressive Party Problem: Integer Linear Programming and Constraint Programming Compared

Barbara M. Smith
Division of Artificial Intelligence, School of Computer Studies,
University of Leeds, Leeds LS2 9JT

Sally C. Brailsford, Peter M. Hubbard and H. Paul Williams
Faculty of Mathematical Studies,
University of Southampton, Southampton SO9 5NH

Abstract

Many discrete optimization problems can be formulated as either integer linear programming problems or constraint satisfaction problems. Although ILP methods appear to be more powerful, sometimes constraint programming can solve these problems more quickly. This paper describes a problem in which the difference in performance between the two approaches was particularly marked, since a solution could not be found using ILP.

The problem arose in the context of organising a "progressive party" at a yachting rally. Some yachts were to be designated hosts; the crews of the remaining yachts would then visit the hosts for six successive half-hour periods. A guest crew could not revisit the same host, and two guest crews could not meet more than once. Additional constraints were imposed by the capacities of the host yachts and the crew sizes of the guests.

Integer linear programming formulations which included all the constraints resulted in very large models, and despite trying several different strategies, all attempts to find a solution failed. Constraint programming was tried instead and solved the problem very quickly, with a little manual assistance. Reasons for the success of constraint programming in this problem are identified and discussed.

1 Introduction

Discrete optimization problems of the kind that arise in many areas of operational research can be formulated as constraint satisfaction problems (CSPs). A CSP consists of a set of variables, each with a finite set of possible values (its domain), and a set of constraints which the values assigned to the variables must satisfy. In a CSP which is also an optimization problem, there is an additional variable representing the objective; each time a solution to the CSP is found, a new constraint is added to ensure that any future solution must have an improved value of the objective, and this continues until the problem becomes infeasible, when the last solution found is known to be optimal.

Many discrete optimization problems can be modeled using linear constraints and integer variables and thus formulated as integer linear programming problems. Operational Research has developed a battery of powerful techniques for solving such problems, but although the search algorithms available for solving CSPs are at first sight less powerful than ILP methods, sometimes constraint programming is a more successful approach (see [1, 4, 5]). It would be very useful to know which of these competing techniques to choose for a given problem, but the boundary between their areas of expertise has not yet been fully mapped. This paper describes a further example of a problem where constraint programming did much better than ILP; in fact, it proved impossible to solve the problem at all using ILP. The success of constraint programming in this case appears to be due to a number of factors in combination; these are discussed in section 8.

The problem is a seemingly frivolous one arising in the context of organising the social programme for a yachting rally. The 39 yachts at the rally were all moored in a marina on the Isle of Wight[1]; their crew sizes ranged from 1 to 7. To allow people to meet as many of the other attendees as possible, an evening party was planned at which some of these boats were to be designated hosts. The crews of the remaining boats would visit the host boats in turn for six successive half-hour periods during the evening. The crew of a host boat would remain on board to act as hosts; the crew of a guest boat would stay together as a unit for the whole evening. A guest crew could not revisit a host boat, and guest crews could not meet more than once. Additional capacity constraints were imposed by the sizes of the boats. The problem facing the rally organiser was that of minimising the number of host boats, since each host had to be supplied with food and other party prerequisites.

There were a number of complicating factors in the real-life problem. For example, the rally organiser's boat was constrained to be a host boat,

[1] off the south coast of England.

although it had a relatively small capacity, because he had to be readily available to deal with emergencies. Two other boats had crews consisting of parents with teenage children, and these boats were also constrained to be host boats; the crews split up so that the parents remained on board the host boat and the children became a "virtual boat" with capacity of zero. The rally organiser's children formed a third virtual boat, giving 42 boats altogether. The data for this problem is given in Table 1.

Boat	Capacity	Crew	Boat	Capacity	Crew	Boat	Capacity	Crew
1	6	2	15	8	3	29	6	2
2	8	2	16	12	6	30	6	4
3	12	2	17	8	2	31	6	2
4	12	2	18	8	2	32	6	2
5	12	4	19	8	4	33	6	2
6	12	4	20	8	2	34	6	2
7	12	4	21	8	4	35	6	2
8	10	1	22	8	5	36	6	2
9	10	2	23	7	4	37	6	4
10	10	2	24	7	4	38	6	5
11	10	2	25	7	2	39	9	7
12	10	3	26	7	2	40	0	2
13	8	4	27	7	4	41	0	3
14	8	2	28	7	5	42	0	4

Table 1: The data

2 The Uncapacitated Problem

If we ignore the capacity constraints, just one host boat can accommodate any number of guest boats for one time period. For more than one time period, we can easily find a lower bound on the number of hosts required from the following argument. If g guest crews visit host i at time 1, then there must be at least g other hosts to accommodate them in the following time period. (The guests cannot visit host i again, and must visit g different hosts so as not to meet another crew again.) In fact, the required $g+1$ hosts could each accommodate up to g visiting guest crews at time 1, without the guest crews meeting again at time 2, giving $g(g+1)$ guest crews in total. For more than 2 time periods, $g(g+1)$ is clearly an upper bound on the number of guest crews that $g+1$ hosts can accommodate. For instance, 6 hosts can accommodate at most 30 guest boats: 7 hosts can accommodate at most 42. In fact, these limits can be attained (still assuming no constraints on the hosts' capacities, and provided that the number of time periods is not greater than the number of hosts, in which case it becomes impossible for guest crews not to visit the same host more than once), so that with 42 boats in all, we need 7 to be hosts (and therefore 35 to be guests).

However, for the real-life problem, the capacity constraints are binding and the number of host boats required is at least 13, as shown in Section 3.

3 A Lower Bound

A lower bound on the number of hosts required, taking into account the capacity constraints, was found by using linear programming to solve a considerable relaxation of the original problem. This simply required that the guest crews, as a whole, should fit into the total spare capacity of the host boats[2] for one time period.

The same lower bound can alternatively be found from a simple argument: a necessary condition for feasibility is that the total capacity of the host boats is not less than the total crew size of all the boats. The smallest number of hosts that meet this condition is therefore found by ordering the boats in descending order of total capacity. With this ordering, the first 13 boats can accommodate all the crews; the first 12 boats cannot.

This suggests that in general the host boats should be chosen in descending order of total capacity. However, this heuristic was not arrived at until after the linear programming work on the problem had been completed, partly because it seemed counter-intuitive that the crew sizes should be ignored when selecting the hosts. Moreover, maximising the number of spare places is not the only consideration when selecting the host boats, since each crew has to stay together. Provided that the total capacity of the hosts is large enough, the choice of hosts may need to consider the spare capacity of each boat and how well different crew sizes fit into it.

Hence the model described below includes the selection of the host boats, even though in practice the choice of hosts was in large part guided by heuristics.

4 Integer Programming Approach

4.1 First Formulation

The first attempt at finding an optimal solution was made at the University of Southampton, where the problem was formulated as a zero-one integer programme. The variables are: $\delta_i = 1$ iff boat i is used as a host boat, and $\gamma_{ikt} = 1$ iff boat k is a guest of boat i in period t. (The rally organiser's boat was constrained to be a host in all models.)

As mentioned in Section 1, the objective was to minimise the number of hosts:

[2] i.e. the remaining capacity after accommodating the host crews themselves.

$$\text{minimise } \sum_i \delta_i \qquad \text{subject to:}$$

Constraints CD. A boat can only be visited if it is a host boat.

$$\gamma_{ikt} - \delta_i \leq 0 \qquad \text{for all } i,k,t; i \neq k$$

Constraints CCAP. The capacity of a host boat cannot be exceeded.

$$\sum_{k,k\neq i} c_k \gamma_{ikt} \leq K_i - c_i \qquad \text{for all } i,t$$

where c_i is the crew size of boat i and K_i is its total capacity.
Constraints GA. Each guest crew must always have a host.

$$\sum_{i,i\neq k} \gamma_{ikt} + \delta_k = 1 \qquad \text{for all } k,t$$

Constraints GB. A guest crew cannot visit a host boat more than once.

$$\sum_t \gamma_{ikt} \leq 1 \qquad \text{for all } i,k; i \neq k$$

Constraints W. Any pair of guests can meet at most once.

$$\gamma_{ikt} + \gamma_{ilt} + \gamma_{jks} + \gamma_{jls} \leq 3 \qquad \text{for all } i,j,k,l,t,s; \; i \neq j; i \neq k; k < l; i \neq l; \; k \neq j; j \neq l; s \neq t$$

The constraints W, which have six indices, clearly lead to a huge number of rows when the problem is modelled. The number of rows is $O(B^4T^2)$, where B is the number of boats and T is the number of time periods. However, this model has a moderate number of variables, namely $O(B^2T)$.

The size of the problem was reduced by taking account of the fact that in any optimal solution there are some boats which will always be chosen to be hosts because of their large capacity and small crew size. By the same token, some boats would clearly never be chosen to be hosts: for example, the three virtual boats with zero capacity. The data was ordered by decreasing (total capacity − crew size) and parameters *hostmin* and *hostmax* were introduced, such that the range of indices for potential hosts was restricted to 1, .. , *hostmax* and the range of indices for potential guests was restricted to *hostmin*+1, .. , 42.

The formulation was tested on a reduced problem with 15 boats and 4 time periods, and with *hostmin* = 4 and *hostmax* = 8. This resulted in a model with 379 variables and 18,212 rows. The LP relaxation solved in 259 seconds using the XPRESSMP optimiser[3] on an IBM 486 DX PC, in 816 simplex iterations.

[3]XPRESS MP (Version 7). Dash Associates, Blisworth House, Blisworth, Northants NN7 3BX, U.K.

4.2 Second Formulation

To reduce the number of constraints in the previous formulation, a further set of zero-one variables was introduced:

$$x_{iklt} = 1 \text{ iff crews } k \text{ and } l \text{ meet on boat } i \text{ in time period } t$$

and the constraints W were replaced by the three following sets S, V and Y. S and V together define the x variables in terms of the γ variables:
Constraints S.

$$2x_{iklt} - \gamma_{ikt} - \gamma_{ilt} \leq 0 \qquad \text{for all } i,k,l,t; k < l; i \neq k; i \neq l$$

Constraints V.

$$\gamma_{ikt} + \gamma_{ilt} - x_{iklt} \leq 1 \qquad \text{for all } i,k,l,t; k < l; i \neq k; i \neq l$$

and constraints Y then replace constraints W in the first formulation:
Constraints Y. Any pair of guest crews can meet at most once.

$$\sum_t \sum_{l,l>k} x_{iklt} \leq 1 \qquad \text{for all } i,k$$

The number of variables is now increased to $O(B^3T)$, but the number of rows is reduced, also to $O(B^3T)$.

5 Experiments on A Reduced Problem

The second formulation was used in a variety of computational experiments with the reduced 15-boat problem. As before, *hostmin* = 4 and *hostmax* = 8. Firstly, the problem was solved directly (model PS1). This gave an optimal solution with 5 hosts in a total time of 2214 secs. This was used as a basis for comparison in several experiments.

First, a facility of the XPRESSMP package was used which enables certain constraints to be introduced only if a particular solution violates them. This greatly reduces the initial size of a model. This facility is called MVUB (Make Variable Upper Bounds) and applies only to constraints of the form $x - My \leq 0$. The CD constraints were in this form already and the S constraints could be disaggregated to get them into the proper form, giving:

$$x_{iklt} - \gamma_{ikt} \leq 0 \text{ and } x_{iklt} - \gamma_{ilt} \leq 0$$

Normally disaggregation would result in a tighter LP relaxation, but since in this case all the coefficients of x_{iklt} in the other constraints of the model are unity, it can be shown by Fourier-Motzkin elimination that this will not

be the case [6]. In the second version of the model (PS11) both the CD and the S constraints were modelled using MVUB; in the third (PS12) the S constraints were modelled explicitly and the CD constraints were modelled using MVUB. The results are shown in Table 2; the total solution time for PS1 was less than for either of the new versions. Thus the MVUB feature was not helpful in this case.

Model	PS1	PS11	PS12
Rows	4386	2130	4022
Columns	2271	2271	2271
LP solution time (secs)	101	16	19
Number of iterations	1474	696	497
LP objective value	3.42	3.42	3.42
MVUB time (secs)	*n.a.*	561	697
Branch-&-Bound time (secs)	2113	1852	3789
Number of nodes	287	279	311
IP objective value	5.00	5.00	5.00

Table 2: Results with MVUB

Next, special ordered sets of type I were tried. A set of variables form a special ordered set of type I if at most one of them can be nonzero. For example, the γ variables could be treated as special ordered sets: for each value of i and k, at most one of the set $\{\gamma_{ikt}, t = 1, .., 6\}$ can be nonzero. This device is useful if there is some natural ordering on the variables, when it can reduce the time spent doing branch-and-bound. However, in this case there is no natural ordering, since the time periods are interchangeable: in any solution, periods 1 and 6, say, can be swapped and the solution will still be valid. This meant that the approach was not helpful and in fact made branch-and-bound slower.

It would also be possible to tighten the LP relaxation by adding extra "covering" constraints generated from the CCAP constraints. For example, from the capacity constraint

$$\gamma_{121} + 2\gamma_{131} + 4\gamma_{141} + 3\gamma_{151} \leq 7$$

the following covering constraints could be derived:

$$\gamma_{121} + \gamma_{141} \leq 1$$

$$\gamma_{121} + \gamma_{131} + \gamma_{151} \leq 2$$

$$\gamma_{121} + \gamma_{141} + \gamma_{151} \leq 2$$

However, there would be a vast number of such constraints and so this did not seem a particularly fruitful approach.

Another approach was to omit the S, V and Y constraints, solve the LP relaxation of the resulting problem and then add in cuts of the form

$$\sum_{i \in Q} \gamma_{kit} - \sum_{i,j \in Q} x_{kijt} \leq \left\lfloor \frac{|Q|}{2} \right\rfloor$$

for index sets Q, where $|Q|$ is odd. By inspection, many of these were violated by the fractional solution. However, automating the process of inspecting the solution, identifying the appropriate index sets and then generating the corresponding cuts would have been computationally prohibitive. Equally, there would be no advantage in generating all possible cuts (even for sets of cardinality 3 only), as this would simply have resulted in another enormous model.

To summarise, the experiments with the reduced problem did not indicate a successful solution strategy for the full problem.

6 Experiments on The Full Problem

The size of the full model defeated all the available modellers, even using indices restricted by *hostmin* and *hostmax*. Therefore, a heuristic approach was adopted, based on the recognition that the total capacity of the first 13 boats (arranged, as described earlier, in decreasing order of spare capacity) was sufficient to accommodate all the crews (there would be 4 spare places), and that the largest guest crew could be accommodated on all but three of the hosts. Hence if a solution with 13 hosts was possible, these 13 hosts seemed a reasonable choice[4].

A solution with 14 hosts was found, by relaxing the meeting constraints and specifying that at least the first 14 boats, and at most the first 15, had to be hosts. An integer solution was found in 598 secs. There were only a few violations of the meeting constraints and, by manually adding in the violated constraints, a feasible solution to the original problem was found.

It began to seem that this might be an optimal solution. Therefore the first 13 boats were fixed as hosts and an attempt was made to prove that this was infeasible. The indices of the guest boats in the meeting constraints were restricted to 21 to 42 (simply because this gave the largest model that XPRESSMP could handle). The model was still large by most standards: 19,470 constraints and 11,664 variables. It was run using a parallel implementation of OSL[5] on seven RS/6000 computers at IBM UK Scientific Centre,

[4]As described in Section 3, it was later realised that the first 13 boats in order of total capacity, K_i, would give a larger number of spare places after all the guests had been accommodated. Nevertheless, the problem was in theory feasible, in terms of total capacity, with the 13 selected hosts.

[5]Optimization Subroutine Library (OSL), IBM Corporation.

Hursley. The run was aborted after 189 hours, having failed to prove infeasibility. OSL had processed about 2,500 nodes, of which around 50% were still active: 239 were infeasible. Some nodes were taking over two hours to evaluate.

7 Constraint Programming Approach

This alternative approach was suggested by the desire to prove infeasibility for the 13-host model, in the light of the failure of OSL. However, it turned out that constraint programming was able to find a feasible solution very rapidly, albeit with manual intervention. The work was carried out at the University of Leeds.

The progressive party problem, with the 13 specified host boats, was formulated as a constraint satisfaction problem (CSP) and implemented in ILOG Solver [3], a constraint programming tool in the form of a C++ library. Solver has a large number of pre-defined constraint classes and provides a standard backtracking search method, the forward checking algorithm, for solving CSPs. If necessary, new constraint classes and search algorithms can be defined, but these facilities were not required in this case. The forward checking algorithm repeatedly chooses an unassigned variable, chooses a value for that variable from its current domain and makes a tentative assignment. The constraints are then used to identify the effects of the assignment on future (still unassigned) variables, and any value in the domain of a future variable which conflicts with the current assignment (and the rest of the current partial solution) is temporarily deleted. If at any stage the domain of a future variable becomes empty, the algorithm backtracks and retracts the last assignment. Solver also makes the problem arc consistent at the outset, and maintains arc consistency in the subproblems consisting of the future variables and their remaining domains, as forward checking proceeds.

7.1 CSP Formulation

The first advantage of the constraint programming approach was that the formulation as a CSP was much more compact than had been previously possible. Since the task in this case was to show (if possible) that the problem with 13 specific host boats was infeasible, host and guest boats were treated separately in the formulation. Suppose that there are G guest boats, H host boats and T time periods.

The principal variables, h_{it}, represent the host boat that guest boat i visits at time t; the domain of each h_{it} is the set $\{1,..,H\}$, and there are GT such variables. The constraints that every guest boat must always have a host and that in any time period a guest boat can only be assigned to one

host are automatically satisfied by any solution to the CSP, which must have exactly one value assigned to each h_{it}, i.e. exactly one host assigned to each guest boat in each time period.

The constraints that no guest boat can visit a host boat more than once are expressed in the CSP by:

$$h_{i1}, h_{i2}, h_{i3}, .., h_{iT} \text{ are all different} \qquad \text{for all } i$$

For each i, this gives a single Solver constraint, equivalent to $T(T-1)/2$ binary not-equals constraints, i.e. $h_{i1} \neq h_{i2}$, etc.

The capacity constraints are dealt with, as in the LP, by introducing new constrained 0-1 variables, corresponding to the γ variables of the LP formulation: $v_{ijt} = 1$ iff guest boat i visits host j at time t. The relationships between these variables and the h_{it} variables are specified by GHT Boolean constraints:

$$v_{ijt} = 1 \text{ iff } h_{it} = j \qquad \text{for all } i, j, t$$

and as in the LP, the capacity constraints are then:

$$\sum_i c_i v_{ijt} \leq C_j \qquad \text{for all } j, t$$

where c_i is the crew size of guest boat i and C_j is the spare capacity of host boat j, after accommodating its own crew.

The constraints that no pair of crews can meet twice also require the introduction of a new set of 0-1 variables: $m_{klt} = 1$ iff crews k and l meet at time t. The constraints linking the new variables to the original h variables are:

$$\text{if } h_{kt} = h_{lt} \text{ then } m_{klt} = 1 \qquad \text{for all } k, l, t; k < l$$

and the meeting constraints are expressed by:

$$\sum_t m_{klt} \leq 1 \text{ for all } k, l; k < l$$

Because the m variables have only three subscripts, rather than four as in the equivalent (x) variables in the LP, the CSP has only $O(B^2T)$ variables and $O(B^2T)$ constraints.

7.2 Symmetry Constraints

The constraints just described are sufficient to define the problem: a number of additional constraints were introduced to reduce the symmetries in

the problem, as much as possible. For any solution, there are many equivalent solutions, which have, for instance, two guest boats with the same size crew interchanged throughout. Such symmetries in the problem can vastly increase the size of the search space and so the amount of work that has to be done. If there are no solutions, searching through many equivalent partial solutions can be extremely time-consuming. Symmetry can be avoided, or at least reduced, by adding constraints to eliminate equivalent solutions. (Puget [2] discusses this approach to avoiding symmetry.)

First, an ordering was imposed on the time-periods, which are otherwise interchangeable: the first guest boat must visit the host boats in order. (As described in the next section, the first guest boat was the one with the largest crew.)

The second set of constraints distinguishes between guest boats with the same size crew: if i, j are such a pair, with $i < j$, then for the first time period, we impose the constraint:

$$h_{i1} \leq h_{j1}$$

To allow for the fact that both boats may visit the same host at time 1 (i.e. $h_{i1} = h_{j1}$), but if so, they must visit different boats at time 2:

$$\text{either } h_{i1} < h_{j1} \text{ or } h_{i2} < h_{j2}$$

Finally, constraints were added to distinguish (to an extent) between host boats with the same spare capacity: if j and k are two such host boats, with $j < k$, the first guest boat cannot visit host k unless it also visits host j.

7.3 Solving the Problem

It is next necessary to choose variable and value ordering heuristics, i.e. rules for deciding which variable to consider next and which value to assign to it. Although the formulation just described has a great many variables, assigning values to the h_{it} variables is sufficient to arrive at a solution, and in devising variable and value ordering heuristics only these principal variables need be considered. Good variable ordering heuristics, in particular, are often crucial to the success of constraint programming methods. A heuristic which is commonly used is based on the "fail-first" principle, that is, choose the variable which is likely to be hardest to assign. In the forward checking algorithm, this means choosing the variable with the smallest remaining domain; ties may be broken by choosing the variable involved in most constraints. Finally, variables are considered in the order in which they are defined; to give priority to the largest crews, the guest crews, and so the corresponding h_{it} variables, were arranged in descending order of size.

In ordering the values, a general principle is to choose first those values which seem most likely to succeed. and the host boats accordingly were considered in descending order of spare capacity.

The problem formulation was first tested on smaller problems than the full 13 hosts, 29 guests, 6 time periods problem. Several smaller problems were solved very quickly (in 1 or 2 seconds on a SPARCstation IPX). with little backtracking, and the program was shown to be producing correct solutions. However, the full size problem ran for hours without producing any result.

It was then decided to assign all the variables relating to one time period before going on to the next. Hence, each time period was solved separately, but the solutions for each time period constrained future solutions. In effect, this was another variable ordering heuristic, taking priority over the others; however, the program was not able to backtrack to earlier time periods. The plan was to find a solution for as many time periods as could be solved within a short time, and then to print out the domains of the remaining variables. It was hoped that this would give some clue as to why the program could not proceed. With this modification, the program found a solution for five time periods very quickly (which it had not previously been able to do). At this point, the domain of any variable corresponding to the 6th time period contains those hosts that the corresponding boat can visit and has not already visited. An attempt was made to fit the guest crews into those host boats which they could still visit, by hand, in order to see why it could not be done. However, a solution was found which appeared to be feasible; adding some of the assignments to the program as extra constraints confirmed that a solution based on these assignments did obey all the constraints.

Hence, the 13 hosts, 29 guests, 6 time periods problem, which had been thought to be insoluble, had been solved, though with some manual assistance. An optimal solution to the original problem had therefore been found.

Subsequently, the extra constraints on the 6th time period were removed, and the program allowed to search for a solution without this intervention; it found a solution in 27 minutes, and went on to find a solution for the 7th time period in another minute, so that the party could have lasted for longer without requiring more hosts!

8 Discussion

For this particular problem, constraint programming succeeded spectacularly in finding a solution very quickly where linear programming had failed to find a solution at all. Moreover, on those problems which linear programming succeeded in solving, constraint programming found solutions much more quickly. A number of reasons to account for the success of constraint programming in this case can be identified.

8.1 Compactness of representation

The fundamental difficulty with linear programming appears to lie in finding a compact representation of the problem. The formulations described earlier show that constraint programming requires far fewer constraints and variables to represent the problem. This is possible because of the greater expressive power of the constraints allowed in constraint satisfaction problems, which are not restricted to linear inequalities as in linear programming. In turn, the greater expressiveness is allowed by the fact that constraint satisfaction algorithms make relatively unsophisticated (although very effective) use of the constraints, compared with the simplex algorithm, for instance; in constraint programming, it is only necessary to be able to detect whether a particular set of assignments satisfies a given constraint or not.

Furthermore, although the total number of variables and constraints is important in constraint programming, it commonly happens, as here, that in modelling a complex combinatorial problem as a CSP, there is a set of principal variables together with subsidiary variables which are introduced for the purpose of modelling the constraints. Typically, the forward checking algorithm is applied only to the principal variables of the problem. As already mentioned, in this case the solution is found by assigning values to the h_{it} variables in turn, i.e. by assigning a host to each guest boat i for each time t. The subsidiary variables are automatically instantiated in this process, because of the constraints linking them to the principal variables, and they are used in effect as vehicles for propagating the problem constraints to the domains of other principal variables. Hence, the effective complexity of the problem is less than the total number of constraints and variables suggests. It is still, however, a large problem, having 29×6 variables, each with 13 possible values, giving $13^{29\times6}$ possible assignments.

8.2 Constraint propagation

The constraints in the progressive party problem are such that, in constraint programming, the effect of any assignment of a value to a variable can usually be propagated immediately to the domains of related variables. For instance, as soon as an assignment is made to an h_{it} variable, i.e. a host is assigned to guest crew i at time t, the same value can be removed from the domain of any variable corresponding to a crew that has already met crew i. The capacity constraints are the only ones that may not immediately prune the domains of other principal variables. However, given the capacities and crew sizes, the maximum number of guest crews that can visit a host simultaneously is 5, and in practice the number is almost always 3 or less. An assignment to an h_{it} variable will, therefore, very often result in the capacity constraints being used to prune other domains. Since the capacity constraints are binding, it is

important that infeasible assignments should be detected as early as possible in this way. In other problems, by contrast, constraints on resources may only have any pruning effect once most of the variables involved in the constraint have been instantiated; this can lead to a large amount of searching before a set of assignments satisfying the constraint is found.

8.3 Solution Strategy

The attempts detailed in section 5 to find a good solution strategy using ILP are based on the mathematical properties of the model and not on the original problem. When it succeeds, this is, of course, one of the strengths of linear programming: the methods that have been developed are independent of the specific problem being addressed. However, in this case, where the attempts failed, it seems a disadvantage that the important features of the original problem cannot be used to direct the search for a solution. Indeed, the ILP model makes it difficult to see that the problem is essentially one of assigning a host boat to each guest boat in each time period: the majority of the variables are x variables, introduced solely to model the meeting constraints, and the fact that each guest boat must be assigned to exactly one host boat at any time is expressed only implicitly in the constraints.

In the CSP formulation, on the other hand, it is easy to see the essentials of the problem, because the principal variables represent precisely the assignment of a host boat to each guest boat in each time period. This allows a solution strategy to be devised around reasoning about these assignments. Although this approach may seem very problem specific, it requires only that variable and value ordering strategies be defined, and often, as here, general principles apply; choose next the variable which is likely to be hardest to assign, and choose a value for it which is likely to succeed. Solving the problem separately for each time period is admittedly a more problem-specific heuristic, but it is an example of an approach worth trying when a problem can be naturally divided into subproblems, and very quick and easy to implement.

8.4 Proof of optimality

Finally, it is easy to show that a solution with 12 hosts is impossible, from the fact that the capacity constraints cannot be met even for a single time period; hence, when a solution with 13 hosts was found, it was known to be optimal. In other situations, proving optimality can be much more difficult.

8.5 The Role of Heuristics

All these factors, together with a degree of good luck in the choice of heuristics, combined to make what was a very difficult problem for linear pro-

gramming, a tractable one for constraint programming. Even so, the size of the problem meant that it was still potentially too difficult for constraint programming to solve if a great deal of search was required.

The role of heuristics in finding a solution is clearly crucial; for instance, the program was unable to find a solution with 13 hosts when all the time periods were considered together. Considering each time period in turn, as it was implemented in this case, has obvious limitations; because backtracking to earlier time periods is not allowed, this strategy is not guaranteed to find a solution if there is one, and it cannot show that a problem is infeasible. In general, it may be necessary to experiment with different combinations of heuristics on a given problem instance in order to get a good solution. Even then, it may be difficult to find a solution: a modification of the original problem, to make the individual crew sizes much more equal while keeping the total size the same, has proved much more difficult (even for one time period, where a solution can easily be found manually).

Ironically, it seems very probable that if the 13-host problem had indeed been infeasible, as originally supposed, the constraint programming approach would not have been able to prove infeasibility: although it is easy to show that a solution with 12 hosts is impossible, because the capacity constraints cannot be met even for a single time period, a problem which is 'only just' infeasible, because the meeting and capacity constraints cannot be simultaneously satisfied for the required number of time periods, would require a complete search of a very large search space, and would be extremely difficult to prove infeasible.

9 Related Work

Papers which also compare integer linear programming and constraint programming applied to particular problems are [1, 5]. Van Hentenryck and Carillon [5] describe a warehouse location problem, and suggest that the ILP model, because it has a great many variables relating to the allocation of customers to warehouses, disguises the fact that the essence of the problem is to decide which of the possible warehouse locations should be chosen. The constraint programming approach, on the other hand, is based on reasoning about the warehouses. This is similar in some respects to the progressive party problem, which also has a large number of additional ILP variables to model the meeting constraints, as described in section 8.3. However, in the warehouse location problem, the ILP and CSP models have an identical set of 0-1 variables representing whether each warehouse is to be used or not, so that the difficulty in the ILP is that the main variables are swamped by other variables. In the progressive party problem, an additional difficulty is that the ILP, unlike the CSP, has no variables representing directly the allocation

of a host boat to each guest boat in each time period.

The paper by Dincbas, Simonis and van Hentenryck [1] discusses a case in which the expressive power of the constraints in constraint programming allows a radical reformulation of the obvious ILP model, giving a much smaller problem. A formulation with n variables, each with m values, and constraints which are linear inequalities, is expressed instead in terms of m variables each with n values, and more complex constraints. This changes the number of possible assignments of values to variables from m^n to n^m. Since the number of possible assignments indicates the total size of the search space, this is an advantage if m is much smaller than n. For instance, in [1], a CSP with complexity 4^{72} is reformulated to give a problem with complexity 72^4.

However, this is not the reason for the success of constraint programming in the progressive party problem: in that case, the CP formulation still has a relatively small number of values compared with the number of variables, and the number of possible assignments is $13^{29\times6}$. In theory, therefore, we should consider reversing the formulation in some way, making the host boats the variables. However, in this case reformulation is not a sensible option. One complication is the time dimension: the variables would have to correspond to each host boat in each time period, giving 13×6 variables in all. The values would then be the possible combinations of guest boats which could be assigned to each variable, and there are a great many such combinations; the constraints would also be very difficult to express. So although reversing the formulation can be extremely valuable in some cases, reducing a large problem to a smaller problem which can be solved much more quickly, it is not possible in this case.

A recent paper by Puget and De Backer [4] compares integer linear programming and constraint programming in general. They conclude that a crucial factor is the degree of propagation that the constraints of a problem allow: if each assignment of a value to a variable can be expected to trigger the pruning of many values from the domains of other variables, so that large parts of the search space do not have to be explored, constraint programming can be expected to be successful. As discussed in section 8.2, the constraints in the progressive party problem are very effective in propagating the effects of assignments to other variables. In other cases, for instance, where the constraints involve large numbers of variables, constraint propagation may be much less useful, and if the problem can be naturally represented by linear constraints, integer linear programming may be more efficient.

10 Conclusions

Although the progressive party problem may not be a practical problem, except for members of yacht clubs, it has many of the classical features of

combinatorial optimization problems, and was expected to be amenable to linear programming techniques. However, as we have shown, the resulting models were too large to be solved, whereas constraint programming found an optimal solution very quickly.

The success of constraint programming in solving this problem is due to a combination of factors, discussed in section 8. Some of these reasons have been identified in other studies of problems where constraint programming out-performed ILP, as discussed in section 9. However, unlike the previous studies, in this problem both models turned out to be extremely large; ILP failed because the model was too large to be solved, but also, it would not have been possible to explore the complete search space arising from the constraint programming formulation. In practice, many real problems are too large to be able to guarantee to find a solution; this paper shows that even so, constraint programming can succeed through careful choice of heuristics to direct its search.

Our experience with this problem suggests that constraint programming may do better than integer linear programming when the following factors are present:

- The problem cannot easily be expressed in terms of linear constraints: constraint programming will then give a more compact representation.
- The constraints allow the early propagation of the effects of assignments to the domains of other variables. This happens if each constraint involves only a small number of variables, but also sometimes with global constraints, as in this case: the capacity constraints are triggered after only a small number of guest boats have been assigned to the same host at the same time.
- It is easy to devise good solution strategies for the problem and hence take advantage of the fact that the constraint programming formulation represents the problem much more directly that the ILP formulation typically does.
- A tight bound on the value of the objective in an optimal solution is available, so that if an optimal solution is found, it can be recognised as such (unless, of course, the problem is sufficiently small to be able to prove optimality by doing a complete search).

Further comparisons between the two approaches are still needed to quantify some of these factors and to give a clearer idea of when constraint programming should be chosen in preference to integer linear programming.

Acknowledgement

We are very grateful to William Ricketts of IBM UK Scientific Centre, Hursley Park, Winchester, for his enthusiastic help with the computational experiments on the large LP model.

References

[1] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving a Cutting-Stock problem in constraint logic programming. In R. Kowalski and K. Brown, editors, *Logic Programming*, pages 42–58. 1988.

[2] J.-F. Puget. On the Satisfiability of Symmetrical Constrained Satisfaction Problems. In *Proceedings of ISMIS'93*, 1993.

[3] J.-F. Puget. A C++ Implementation of CLP. In *Proceedings of SPICIS94 (Singapore International Conference on Intelligent Systems)*, 1994.

[4] J.-F. Puget and B. De Backer. Comparing Constraint Programming and MILP. Submitted to the 1st International Joint Workshop on Artificial Intelligence and Operations Research, June 1995.

[5] P. van Hentenryck and J.-P. Carillon. Generality versus Specificity: an Experience with AI and OR techniques. In *Proceedings of AAAI-88*, volume 2, pages 660–664, 1988.

[6] H. P. Williams. The elimination of integer variables. *JORS*, 43:387–393, 1992.

The TSP Phase Transition

Ian P. Gent
Department of Computer Science
University of Strathclyde
Glasgow G1 1XH
United Kingdom
ipg@cs.strath.ac.uk

Toby Walsh
Mechanized Reasoning Group
IRST, Trento &
DIST, University of Genoa,
Italy
toby@irst.it

In the first instance, please address correspondence to Toby Walsh at IRST, Location Panté di Povo, I38100 Trento, Italy.

Abstract

We wish to bring to the attention of the OR community the phenomenon of phase transitions in randomly generated problems. These are of considerable practical use for benchmarking algorithms. They also offer insight into problem hardness and algorithm performance. Whilst phase transition experiments are frequently performed by AI researchers, such experiments do not appear to be in common use in the OR community. To illustrate the value of such experiments, we examine a typical OR problem, the traveling salesman problem. We report in detail many features of the phase transition in this problem, and show how some of these features are also seen in real problems.

Acknowledgements

The second author is supported by a HCM Postdoctoral Fellowship. We thank Iain Buchanan for comments on a draft of this paper, and Alan Bundy, and the members of the Mathematical Reasoning Group in Edinburgh for their constructive comments and many CPU cycles donated to these and other experiments from SERC grant GR/H/23610. We also thank the MRG group at Trento and the Department of Computer Science at the University of Strathclyde for additional CPU cycles. Finally, we thank Robert Craig for providing us with his code.

1 Introduction

A perennial problem for those wishing to study algorithms is a fair means of comparison. The set of benchmark problems is often comparatively small, and it is hard to be sure that good performance is not the result of luck. Formal theorems on performance are solid, but are often hard to come by and may not be relevant to practical problems. Algorithms can, of course, be tested on random problems, but there is justifiable suspicion of such results, as random problems are not meaningful in themselves. We wish to draw the attention of the OR community to the phenomenon of "phase transitions" in randomly generated problems. This phenomenon allows us to generate random problems which are typically hard, and therefore provide a fair basis for comparison of different algorithms. They can also provide a basis for furthering our understanding of the way algorithms behave on both random and real problems. We hope therefore to contribute to the developing "empirical science of algorithms" [10].

Phase transitions have received considerable attention in the AI community [2, 13, 7]. Whilst random problems are typically easy to solve, hard random problems can be found at a phase transition [2]. AI researchers now routinely use such problems to benchmark satisfiability and constraint satisfaction algorithms. Simple scaling laws are often associated with these phase transitions. For example, scaling laws have been observed both for properties of random problems like the probability of having a solution [12, 7], and for properties of algorithms like the fitness of solutions during hill-climbing [3]. Such scaling laws are likely to prove useful in theoretical analyses of problem hardness and algorithm performance.

In this paper, we show how phase transition phenomena are of practical use in studying a typical OR problem, the traveling salesman problem (TSP). We start by showing that, contrary to earlier reports, there is a very clearly marked phase transition in TSP. We observe an "easy-hard-easy" pattern in median problem difficulty, the hard region being correlated with a change from soluble to insoluble problems. We then show empirically that a simple scaling law holds of this phase transition. We next investigate the occurrence of rare exceptionally hard problems, which can be orders of magnitude harder than those found directly at the solubility phase transition. We show that their hardness is due to early mistakes in backtracking search. Finally, we show that we can study phase transitions in real problems, and that many of the features observed in random problems occur in real problems too.

2 Computational Phase Transitions

Randomly generated problems usually have a natural order parameter. For example, for randomly generated graph colouring problems, the order parameter is the average connectivity of the graph. A rapid phase transition from colourable to uncolourable occurs at a fixed value of this order parameter. Surprisingly this

value is almost completely independent of the size of the graph. Computational hardness appears to be associated with this phase transition [2]. In the colourable region, graphs have low connectivity. As almost any assignment of colours to nodes is a proper colouring, such graphs are usually easy to colour. In the uncolourable region, graphs have high connectivity. As many nodes are connected together, it is usually easy to show that there is an insufficient number of colours with which to colour the graph. By comparison, problems from the phase transition are typically hard since they are neither easily colourable nor obviously uncolourable.

Similar phase transition behaviour has been observed with many different randomly generated NP-complete problems using a variety of different complete and incomplete algorithms: for example, it has been seen in randomly generated satisfiability problems [2, 13], independent set problems [6], Hamiltonian circuits [2], and constraint satisfaction problems [14]. Phase transition phenomena have also been observed in real computational data like exam time-tabling [4]. Given this large spectrum of problem types and algorithms, computational hardness in AI is now often associated with the occurrence of a phase transition. The aim of this paper is to demonstrate the existence of such phase transition behaviour in a typical OR problem.

3 Random TSP problems

Given n cities, a tour length l and a matrix that defines the distance between each pair of cities, the traveling salesman decision problem (the TSP decision problem) is to determine if a tour of length l or less exists which visits all n cities. The TSP decision problem is one of the most famous NP-complete problems. Randomly generated TSP problems can be easily constructed by placing cities on a square of area A at random.

To solve the TSP decision problem, we first use an implementation of the branch and bound algorithm written by Robert Craig at AT&T Bell Labs. This uses the Hungarian algorithm for branching, minimally adapted by us to solve the decision rather than minization problem. Branch and bound is one of the best complete algorithms for the TSP decision problem.

For random TSP problems with n cities uniformly distributed over a rectangular area A, the expected optimal tour length [1] is,

$$l_{opt} = k.\sqrt{n.A} \tag{1}$$

where $k \approx 0.75$. This would suggest that a natural order parameter for the TSP decision problem is $l/\sqrt{n.A}$. Irrespective of the actual values for l, n and A, we expect a phase transition to occur in the probability of a tour existing for a value of the order parameter of about 0.75.

In Figure 1, we plot one view of the phase transition as we vary this order parameter. The number of randomly generated cities is fixed at 24 and the square side is fixed at 1000 units. We vary the tour length required, using the same

1000 sets of 24 random cities at each point. We plot results by means of contours of percentiles, where for example the 50% contour is the median, and the 99% contour represents behaviour that was only exceeded by 1% of problems (in this case 10 problems). We also include the best and worst case. As search cost varies over many orders of magnitude. we plot the *log* of the number of nodes searched. All logs in this paper are to base 10.

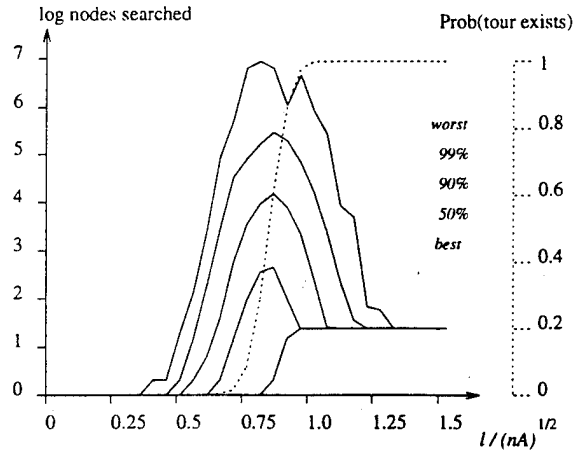


Fig 1: Varying tour length

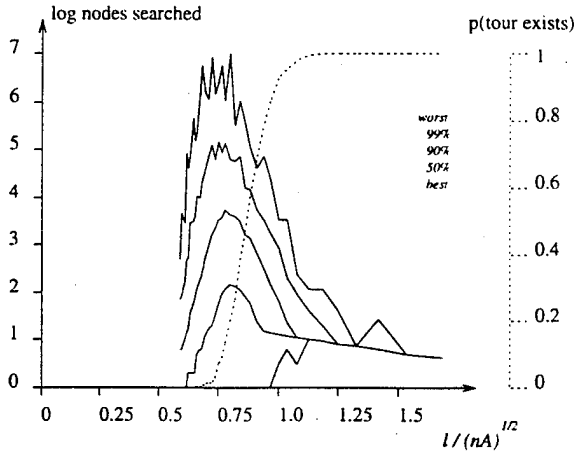


Fig 2: Varying number of cities

As with graph colouring, there is a rapid phase transition from soluble to insoluble as we vary the order parameter. The phase transition occurs, as expected from Equation 1, in the region around $l/\sqrt{n.A} \approx 0.75$. In the soluble region, problems are under-constrained and typically easy. As many tours are less than the required length, it is not difficult to find a tour that is short enough. In the insoluble region, problems are over-constrained and again typically easy. As the required tour length is a very tight bound, many tours are too long and are quickly ruled out. In the phase transition inbetween, problems are "critically constrained" and typically hard. In this region, it is difficult to determine if a tour of the required length exists without exhaustive search. This pattern of "easy-hard-easy" behaviour in the median contour is familiar from AI research into phase transitions [13]. Figure 1 clearly refutes the claim of Kirkpatrick and Selman [12] that "there are other NP-complete problems (for example, the traveling salesman problem or max-clique) that lack a clear phase boundary at which 'hard problems' cluster".

Note that whilst median problem difficulty peaks in the middle of phase transition, occasional very hard problems occur in under-constrained regions where almost all problems are soluble. The worst such problem took 4,412,760 nodes in a region where 97.7% of problems (including this one) have a tour. This is four orders of magnitude worse than the worst median problem, which took just 449 nodes in a region where 61.2% of problems have a tour. Such behaviour has previously been observed both for random graph-colouring and satisfiability problems [9, 5]. We return to such occasional hard problems in satisfiable regions in §5 and

§6.

We observe a similar phase transition when we fix the tour length and vary the number of cities visited. In Figure 2, we set $l/\sqrt{A}$ to $\frac{3}{4}.\sqrt{25}$ and vary the number of cities visited from 5 to 40 in steps of 1, testing 1000 sets of cities at each point. As expected from Equation 1, the phase transition occurs broadly around the region $l/\sqrt{n.A} \approx 0.75$. Median behaviour displayed the usual easy-hard-easy pattern through the phase transition. Compared to Figure 1, the worst case plot shows worse behaviour at small values of the order parameter, in the mostly insoluble region. Exceptionally hard problems in the mostly soluble region do not seem to occur. However, we note that as the order parameter varies, so does the number of cities. Problems in the soluble region have smaller numbers of cities and so are much easier. The worst case is bounded by a search envelope of size $O(n!)$ which is constant in Figure 1 but decreasing to the right in Figure 2. The total number of nodes searched is thus not a direct measure of *relative* problem hardness.

4 Scaling of Phase Transition

To utilise phase transition phenomena effectively, we need to determine how they scale with problem size. For instance, to test heuristic procedures like simulated annealing on random TSP problems which are larger than can be solved using complete procedures, we need a good estimate of the scaling of the probability that a random TSP problem has a tour. Analogies with phase transitions in physical systems are often useful in discovering such scaling results.

One of the most unusual and theoretically interesting phase transitions in physics occurs in spin glasses. Kirkpatrick *et al.* [11] have used an analogy with this phase transition to suggest a simple scaling result for the probability of satisfiability for a common class of randomly generated satisfiability problems. In [7], we demonstrated that this simple scaling result applied to a wide range of different classes of randomly generated satisfiability problems. Properties of algorithms like the number of constraint propagations performed on each branch of the search tree also appear to obey similar scaling results [8].

In a spin glass, each atom has a magnetic spin which can have only one of two values, "up" or "down" (1 or -1). The system therefore has a exponentially large number of possible configurations. Interactions between atoms are both ferromagnetic (promoting alignment of spins) and anti-ferromagnetic (promoting opposite spins). As a result, a spin glass is a "frustrated" system with a large number of near optimal equilibrium configurations which cannot be locally improved by flipping a spin. It is difficult therefore to get the spin glass into a state of least energy. An analogy can be made with TSP problems. A TSP problem has an exponentially large number of possible tours. It is also usually a frustrated system, having a large number of near optimal tours which cannot be locally improved (*eg.* using local changes like 2-opt or 3-opt). It is very difficult therefore to find an optimal tour. This analogy with spin glasses suggests that a macroscopic property like the

probability of a tour existing obeys the following simple scaling law,

$$Prob(tour\ exists) = f((\frac{l}{\sqrt{n.A}} - \alpha).n^{1/v}) \qquad (2)$$

where f is some fundamental function, and α and v are constants.

To test this hypothesis, we generated random TSP problems with $n = 9$, 16, 25, and 36, and measured the probability that a tour exists as we traverse the phase transition. In Figure 3, each probability curve is plotted against l. The solid line represents $n = 36$. As n increases, the phase transition occurs at larger l but over a smaller range. By rescaling the x-axis for each problem size, we can quickly test (2). In line with (1), we set $\alpha = 0.75$. We varied v, and observed the best fit to (2) at $v = 2$. Figure 4 shows the data from Figure 3 rescaled so that the probability of a tour existing of length l for a random n-city tour is plotted at the x-ordinate $(\frac{l}{\sqrt{n.A}} - 0.75).n^{1/2}$. Equation (2) is obeyed if all the probability curves coincide. Figure 4 shows that the data fits (2) very well indeed. Note that (1) is only an asymptotic limit. Using Figure 4, we can for the first time predict the probability of a tour of a given length existing for *all* values of l, n, and A.

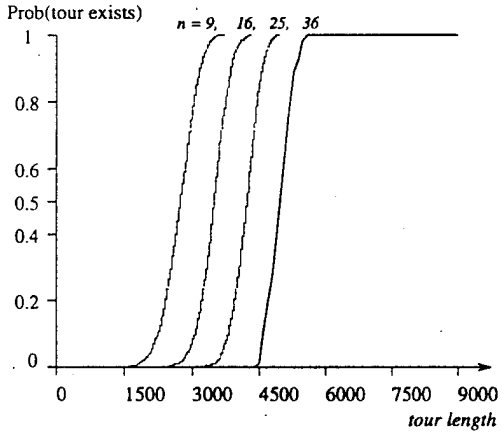


Fig 3: Prob(tour) as n varies

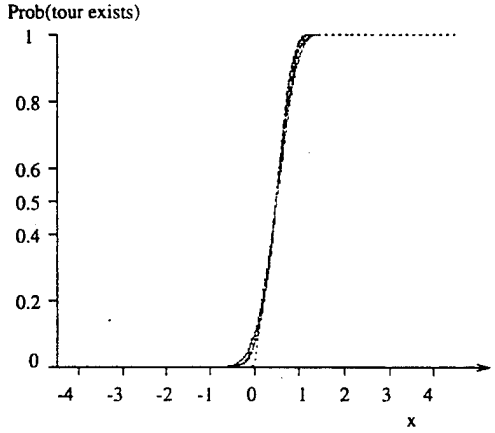


Fig 4: Rescaled data (see text)

Our results very strongly suggest that the probability of a tour existing obeys (2) with the parameters $\alpha \approx 0.75$, $v \approx 2$. So far, we have computed values for f by experimentation. Finding a closed form for f remains an interesting open problem.

5 Optimal tours

The phase transition in Figure 1 results from a mixture of behaviours on soluble and insoluble problems. To isolate the different behaviour, we found the optimal tour for the first 100 sets of cities tested in Figure 1. We could then set the tour length required to be a known distance, d from the optimal tour for each problem.

At $d = 0$ only optimal tours will be found whilst at $d = -1$ no tours will be found at all. By design, the phase transition thus occurs abruptly between $d = -1$ and 0 (indicated by the dashed line). This gives a clear picture of how problem difficulty is related to the distance from the individual transition in solubility. In Figure 5, we vary d, the distance from the optimal tour from -1000 (insoluble) to $+1000$ (soluble) in steps of 50, and for finer detail from -50 to $+50$ in steps of 5. In addition, we plot the cost at -1. For each distance d, we tested 100 sets of capitals, with $l = l_{opt} + d$.

Figure 5 demonstrates two different types of behaviour. In the insoluble region, there is exponential growth as the phase boundary is approached. In the soluble region, problems typically become harder as we approach the phase boundary since we can accept increasingly fewer sub-optimal tours. However, some of the hardest soluble problems have regions where search increases exponentially as we move *away* from the phase boundary. With these problems, poor branching decisions early on result in an exhaustive (and unsuccessful search) for a tour of given length: Naïvely, accepting longer tours should make soluble problems easier, but Figure 5 clearly shows that the opposite can happen.

As an example, the worst case behaviour of 5,011,786 branches occurs at $l_{opt} + 750$, well away from the phase transition, in a region where over 90% of the problems are trivial. This worst case behaviour is a result of poor initial branching decisions followed by an inability of branch and bound to cut off search until a very late stage. In this case, the sixth branching decision made by the Hungarian heuristic was to go from city 1 to city 14, thereby missing out an essential side-trip between the two. In addition, the bound did not cut off search in this blind alley until only two branching decisions were left. At the optimal tour length, by comparison, the very tight bound enables backtracking to be initiated much earlier. The correct tour is thus found in just 22,129 branches. Sudden drops in difficulty in the worst case contour follow the appearance of new and significantly longer sub-optimal tours.

Problems such as the worst case in Figure 5 are often called "exceptionally hard problems". They are of great interest as they appear *away* from the phase transition in satisfiable regions where almost all problems are very easy [9, 5]. Poor early branching choices followed by the inability to cut off search early has also been proposed as the source of exceptionally hard problems in both satisfiability [5, 8] and constraint satisfaction [16]. It remains an open question whether this behaviour is seen with all algorithms or is restricted to those like back-tracking which commit to early decisions. These results demonstrate that, contrary to conventional wisdom, the hardest problems can be *soluble*.

6 Real TSP problems

Random problems may, of course, not be representative of the problems met in practice. For example, they may lack features that can make real problems very

hard. With time-tabling problems, we discovered that problem difficulty can be strongly influenced by the presence of large exam cliques [4]. There large scale structures were rare in our randomly generated time-tabling problems but present in our real data. Such large scale structures can make problems hard even well away from the phase transition. Phase transitions in randomly generated data should therefore be compared with phase transitions in real data.

As an example, we take some standard benchmark data from TSPLIB [15], the capitals of the 48 contiguous states of the U.S.A. To compute the order parameter for this data, we take A to be 7,825,118 km^2, the area of the 48 contiguous states. In Figure 6, we plot one view of a phase transition in this real data as we vary this order parameter, again using the branch and bound algorithm. We fix the number of capitals n at 24 and vary the tour length l from 500km to 15,000km in steps of 500km. We used the same 1000 sets of 24 capitals at each point. A rapid phase transition can be seen in Figure 6 at around $l \approx 8000$km and $l/\sqrt{n.A} \approx 0.6$. The worst case needed 2,933,071,577 nodes for a tour of length 8500km or less, in a region where 99.6% of the problems have a tour, and more than 1446km longer than the optimal tour for this problem. This is over *five* orders of magnitude worse than the worst median of 5,096 nodes at tour length 7500km, where 31.1% of tours were possible and worst case behaviour was 232,077,515 nodes. A similar phase transition occurs when the tour length is fixed and the number of capitals visited is varied [4].

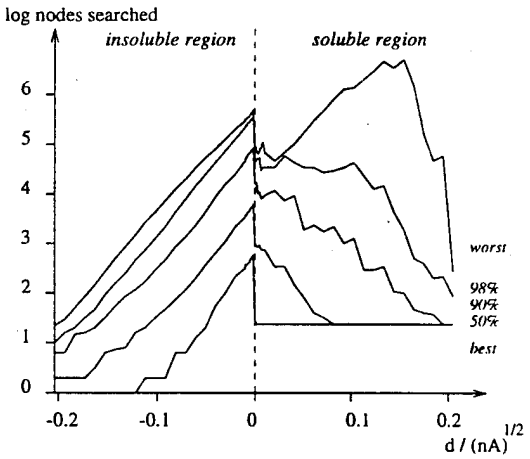


Fig 5: $l_{opt} + d$ for random problems

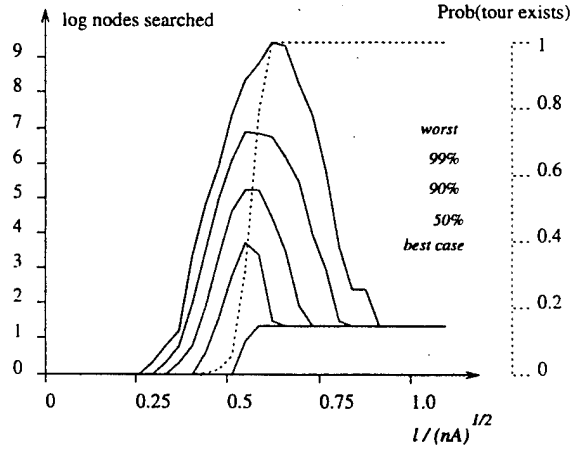


Fig 6: Varying tour length for real problems

The phase transition for real data is very similar to that seen with random data. Median problem difficulty again follows an easy-hard-easy pattern through the phase transition, whilst exceptionally hard problems occur in an under-constrained region where nearly 100% of problems are soluble. The major difference between real and random data is that real TSP decision problems are significantly harder than random ones. In [4], we conjecture that this may be because of the different distribution of cities in randomly generated problems compared to the real data. Exceptionally hard problems in satisfiable regions appear for similar reasons to

random data: an early incorrect branching decision followed by the bound not cutting off search until very deep in the search tree.

7 Conclusions

Phase transition phenomena have been of considerable practical value in AI but have yet to receive much attention in OR. In this paper we have shown that for complete procedures like branch and bound applied to the traveling salesman problem, median problem difficulty typically follows an easy-hard-easy pattern through the phase transition. Exceptionally hard problems can, however, occur in under-constrained regions where nearly 100% of problems are soluble.

We have shown that phase transitions can be seen with both real and random data. Real data is, however, often significantly harder than random data. Phase transitions in real data are thus a very good source of hard problems for benchmarking algorithms. In addition to locating the really hard problems, phase transition phenomena obey some fascinating scaling laws, and we have empirically demonstrated one such law in this paper. Such scaling results promise to improve our understanding of both problem hardness and algorithm performance. As a consequence, phase transition experiments have an important role to play in both AI and OR.

References

[1] J. Beardwood, J.H. Halton, and J.M. Hammersley. The shortest path through many points. *Proceedings of the Cambridge Philosophical Society*, 55:299–327, 1959.

[2] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Proceedings of the 12th IJCAI*, pages 331–337. International Joint Conference on Artificial Intelligence, 1991.

[3] Ian P. Gent and Toby Walsh. An empirical analysis of search in GSAT. *Journal of Artificial Intelligence Research*, 1:47–59, September 1993.

[4] Ian P. Gent and Toby Walsh. Computational phase transitions in real problems. Research Paper 724, Dept. of Artificial Intelligence, Edinburgh, 1994. Submitted to IJCAI-95.

[5] Ian P. Gent and Toby Walsh. Easy problems are sometimes hard. *Artificial Intelligence*, 70:335–345, 1994.

[6] Ian P. Gent and Toby Walsh. The hardest random SAT problems. In Bernhard Nebel and Leonie Dreschler-Fischer, editors, *KI-94: Advances in Artificial Intelligence. 18th German Annual Conference on Artificial Intelligence*, pages 355–366. Springer-Verlag, 1994.

[7] Ian P. Gent and Toby Walsh. The SAT phase transition. In *Proceedings of ECAI-94*, pages 105–109, 1994.

[8] Ian P. Gent and Toby Walsh. The satisfiability constraint gap. Research Paper 702, Dept. of Artificial Intelligence, Edinburgh, June 1994. Revised version to appear in *Artificial Intelligence* special issue on phase transitions in problem spaces.

[9] T. Hogg and C. Williams. The hardest constraint problems: A double phase transition. *Artificial Intelligence*, 69:359–377, 1994.

[10] J. N. Hooker. Needed: An empirical science of algorithms. *Operations Research*, 42:201–212, 1994.

[11] S. Kirkpatrick, G. Györgyi, N. Tishby, and L. Troyansky. The statistical mechanics of k-satisfaction. *Advances in Neural Information Processing Systems*, 6, 1993.

[12] S. Kirkpatrick and B. Selman. Critical behavior in the satisfiability of random boolean expressions. *Science*, May 27 1994.

[13] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of SAT problems. In *Proceedings, 10th National Conference on Artificial Intelligence.* AAAI Press/The MIT Press, July 12-16 1992.

[14] Patrick Prosser. Binary constraint satisfaction problems: Some are harder than others. In *Proceedings of ECAI-94*, pages 95–99, 1994.

[15] G. Reinelt. TSPLIB - a traveling salesman problem library. *ORSA Journal on Computing*, 3:376–384, 1991.

[16] Barbara M Smith and Stuart A. Grant. Sparse constraint graphs and exceptionally hard problems. Research Report 94.36, School of Computer Studies, University of Leeds, 1994.

Extended Abstract:

Integer Programming for Job Shop Scheduling and a Related Problem

E. Andrew Boyd[1]

Texas A&M University

January, 1994

[1]Work performed jointly with Rusty Burlingame, Texas A&M University. However, only Dr. Boyd is applying to attend the conference.

1 Introduction

The job shop scheduling problem is simple to state. Given m machines that can process only one job at a time, and given n jobs that must be processed on each of these machines in a prescribed, job-dependent order, find a scheduling of jobs to machines that minimizes the time the last job is completed. Processing times can vary from job to job and from machine to machine. The most striking aspect of this problem, and the aspect that makes it an interesting topic for a conference on the relationship between OR and AI techniques, is that it is a relatively easy problem to generate good feasible solutions for but a painfully difficult problem to solve to provable optimality.

In 1965 Muth and Thompson proposed a 10 job, 10 machine problem that remained unsolved for two decades. During this period, many researchers developed and applied new algorithmic ideas for solving the job shop problem, including Ashour and Hiremath [4], Balas [5], Barker and McMahon [7], Fisher, Lageweg, Lenstra, and Rinnooy Kan [13], and many others. However, it was not until 1985 that Carlier and Pinson [11] finally published the first optimal solution to the problem.

Following 1985 a number of papers were published on the polyhedral structure of scheduling polyhedra, including Balas [6] and Dyer and Wolsey [12], and in 1991 Applegate and Cook [3] undertook an extensive computational study based on this work. The results of this excellent paper demonstrated that existing polyhedral results are completely inadequate for solving job shop scheduling problems. Applegate and Cook concluded their study by enhancing the primarily combinatorial algorithm of Carlier and Pinson, solving some difficult 10 job, 10 machine problems in the best known times to date. They remained un-

able to solve 7 difficult test problems from the literature, containing up to 15 machines and 20 jobs, and presented a challenge to the optimization community to solve these problems.

Recognizing the proven difficulty of solving job shop scheduling problems, many studies have been undertaken using genetic algorithms, simulated annealing, tabu search, and other intelligent search procedures, including work by Adams, Balas, and Zawack [2], van Laarhoven, Aarts, and Lenstra [20], Aarts, van Laarhoven, Lenstra, and Ulder [1], Taillard [19], Barnes and Chambers [8], and others. In many cases the best known feasible solutions for problems from the literature have been found using these algorithms. Of course, these solutions are not known to be optimal since heuristic algorithms do not provide a proof of optimality.

In our research we have developed an algorithm for solving job shop scheduling problems optimally using a combination of new algorithmic ideas and one of the most recent parallel computer architectures. The algorithm provides the best known performance on a collection of standard test problems from the literature on a scalar platform. However, we still have many enhancements designed for the algorithm, and our goal is to solve some of the very difficult open problems posed by Applegate and Cook. We hope to present new records at the conference, but are in the process of completing the enhancements and the port to the parallel platform, so we will not have any definitive results for at least another month.

In an effort to conform to the agenda of the conference, the proposed presentation will consist of three parts. The first part will discuss general difficulties encountered by exact solution methods for scheduling problems while describing important aspects of the algorithm we have developed. The second part will present computational results for some

famous test problems, presenting both our new results and the best published results, and drawing from both the the OR and AI literature. The final part will briefly present results for a mathematically related problem arising in the context of national air traffic management, in which very large problem instances were solved to provable optimality very quickly. In this problem, the "jobs" are planes and the "machines" are airports. We will comment on what makes one problem class so easy and the other so difficult.

As a personal note, I am primarily an optimizer and believe in the use of optimization techniques whenever appropriate. However, as I continue to gain access to more problems that come to me from various industrial sources, it is increasingly clear there are many important problems that are not amenable to exact solution procedures. I would like to come to better understand many of the intelligent search procedures now being employed in the AI community, and the conference would provide an opportunity for me to achieve this.

2 The Algorithm

All exact methods for the job shop scheduling problem are based on semi-enumerative branch-and-bound algorithms. The primary obstacle to the success of these algorithms on larger job shop scheduling problems is the lack of a good bounding procedure, leading to extremely large tree searches. The method of bounding used most commonly in optimization contexts — formulating the problem as an integer program and solving the linear programming relaxation — has thus far proven inadequate due to the weakness of the integer programming formulations that have been used. The work of Carlier and Pinson and

of Applegate and Cook focuses on using logical conditions to permanently fix variables at optimal values, and using bounds from single machine subproblems that incorporate these fixed variables. While the bounds are relatively weak, a significantly better alternative has not been proposed, and what the the bounds lack in strength they make up for in speed of computation.

Our computational work has proceeded in two different directions. In the first direction, we have enhanced the earlier work of Carlier and Pinson and of Applegate and Cook, introducing new methods of variable fixing and extensions to the bounding procedure. We have also paid careful attention to developing an algorithm that would parallelize efficiently. The computational results presented in the following section were achieved using this algorithm.

The second research direction represents a more radical departure from existing methodologies and is based on relatively recent polyhedral results developed by Dyer and Wolsey [12]. The computational work of Applegate and Cook demonstrated that with proper polyhedral refinements, the standard integer programming formulation of the job shop scheduling problem could be strengthened so that the bounds were roughly equivalent to those obtained by the combinatorial bounding procedure used by Carlier and Pinson. The polyhedral bounds could actually be made slightly stronger but at a very high computational price. The strength/time tradeoff was very poor, and Applegate and Cook rightly concluded that a polyhedral algorithm based on the standard formulation and known results would not prove successful.

What has not been explored computationally, however, is the use of a complete *reformulation* of the problem. Reformulation methods differ from polyhedral methods in that

the formulation is completely rethought rather than refined, and this is often achieved by defining new variables. (The polyhedral relationship between different formulations is itself an interesting problem that has received some attention: see for example Boyd [9], Martin [15], Rardin and Wolsey [18], and others.) The resultant reformulations often have linear programming relaxations that generate much better bounds than their counterparts. Unfortunately, these better bounds are commonly accompanied by a much larger problem formulation.

In the case of job shop scheduling with integral processing times it is possible to define the problem using binary variables t_{ijk}, where t_{ijk} is 1 if job j is scheduled to start on machine i at time k. Dyer and Wolsey [12] used a formulation based on these variables for a single machine scheduling problem, demonstrating that the bounds generated by this formulation are often very good, a fact generally conceded by most integer programmers. The difficulty, of course, is that this problem formulation is generally quite large. For example, with 15 jobs, 15 machines, and 200 potential time periods for each job/machine pair (a reasonable estimate when a good feasible solution for the problem is known and obvious limits on starting times are taken into account), a complete problem reformulation would involve some 45,000 variables. Although solving problems of this size is easily possible with present linear programming techniques, the formulations also have a similar number of constraints and the constraints themselves are relatively dense.

It is conceivable but not likely that an algorithm based on this reformulation could successfully solve any unsolved problems. However, it is possible to exploit the strength of this reformulation to generate good lower bounds for small subproblems in a branch-and-

bound algorithm for the standard formulation. The bounds generated by this reformulation are often significantly stronger than those generated by the combinatorial procedure used by Carlier and Pinson. Even more, this reformulation is not limited to a single machine, and so is not limited by single machine bounds used in all previous work. Bounds generated from more than one machine will almost certainly prove vital in solving larger problems. Finally, since the formulation will be used for generating lower bounds via a linear programming relaxation, if the dual is solved good lower bounds can be generated without the need to solve the relaxation to optimality. Thus, it should be possible to use even relatively large subproblems for bound generation.

It is important to realize that even minimally stronger bounds hold the key to solving larger job shop scheduling problems. The problems that are now considered difficult require millions and often tens of millions of nodes in the branch-and-bound tree. As much time as these problems take to solve, memory usage is often the more significant issue. On difficult problems, algorithms based on existing bounding procedures often fail to yield *any* improvement for tens of thousands of nodes. If they can be produced in a reasonable length of time, even marginally stronger bounds will have a profound impact on the size of problems that can be solved.

3 Computational Results to Date

Computational results obtained to date on the set of 10 by 10 problems considered by Applegate and Cook are presented in Table 1, with values from the Applegate and Cook paper presented for comparison. While in all but one case these results are the best known

for these problems, they represent our present state of development and not our ultimate goal. Further enhancements to the algorithm remain to be made, as does a port to a parallel platform, the Silicon Graphics Power Challenge. We hope (and expect) to be able to report on the solution of some of the open problems for the conference.

			Burlingame and Boyd		Applegate and Cook	
Problem	Init. Feas.	Optimal	Time (Seconds)	Nodes	Time (Seconds)	Nodes
MT10	930	930	113.1	6399	314.8	16055
ABZ5	1245	1234	816.0	52781	837.8	57848
ABZ6	943	943	16.9	985	22.9	1269
LA19	848	842	417.6	26613	1300.7	93807
LA20	911	902	1144.6	74553	1269.1	81918
ORB1	1070	1059	370.9	18075	1379.5	71812
ORB2	890	888	844.6	48447	2290.5	153578
ORB3	1021	1005	3958.8	240747	2159.8	130181
ORB4	1019	1005	140.7	7007	935.2	44547
ORB5	896	887	125.4	7949	398.0	23113

For purposes of comparison, the value of the initial feasible solution was taken from the paper of Applegate and Cook and the heuristic time required to find this solution is not included (the heuristic represented an average of about 15% of total running time for the Applegate and Cook results). We have focused on bounding and variable fixing, but use a simple depth first branching strategy. Thus, when the initial feasible solution is too far from optimality the iteration count potentially can be severely effected, and this accounts for the behavior of the algorithm on problem ORB3. For example, when an initial feasible solution of 1006 is used for ORB3, the node count reduces to 47829 and the time to 818.8 seconds.

Table 1: Results for 10 by 10 Problems

References

[1] Aarts, E. H. L., P. J. M. van Laarhoven, J. K. Lenstra, and N. L. J. Ulder. 1994. A Computational Study of Local Search Algorithms for Job Shop Scheduling. *ORSA Journal on Computing* **6**, 118-125.

[2] Adams, J., E. Balas, D. Zawack. 1988. The Shifting Bottleneck Procedure for Job Shop Scheduling. *Management Science* **34**, 391-401.

[3] Applegate, D., and W. Cook. 1991. A Computational Study of the Job-Shop Scheduling Problem. *ORSA Journal on Computing* **3**, 149-156.

[4] Ashour, S. and S. R. Hiremath. 1973. A Branch-and-Bound Approach to the Job-Shop Scheduling Problem. *International Journal of Production Research* **11**, 391-401.

[5] Balas, E. 1969. Machine Sequencing via Disjunctive Graphs: An Implicit Enumeration Algorithm. *Operations Research* **17**, 941-957.

[6] Balas, E. 1985. On the Facial Structure of Scheduling Polyhedra. *Mathematical Programming Study* **24**, 179-218.

[7] Barker, J. R. and McMahon. 1985. Scheduling the General Job-Shop. *Management Science* **31**, 594-598.

[8] Barnes, J. W. and J. Chambers. 1992. Solving the Job Shop Scheduling Problem Using Tabu Search. Technical Report ORP91-06, Graduate Program in Operations Research, University of Texas at Austin.

[9] Boyd, E. A. 1992. A Pseudopolynomial Network Flow Formulation for Exact Knapsack Separation. *Networks* **22**, 503-514.

[10] Burlingame, R., E. A. Boyd, and K. S. Lindsay. 1993. Solving Large Integer Programs Arising from Air Traffic Control Problems. *Air Traffic Control Quarterly*, **1**, 255-276.

[11] Carlier, J. and E. Pinson. 1989. An Algorithm for Solving the Job-Shop Problem. *Management Science* **35**, 164-176.

[12] Dyer, M. and L. A. Wolsey. 1990. Formulating the Single Machine Sequencing Problem with Release Dates as a Mixed Integer Program. *Discrete Applied Mathematics* **26**, 255-270.

[13] Fisher, M. L., B. J. Lageweg, J. K. Lenstra, and A. H. G. Rinnooy Kan. 1983. Surrogate Duality Relaxation for Job Shop Scheduling. *Discrete Applied Mathematics* **5**, 65-67.

[14] Liu, W. G. 1988. Extended Formulations and Polyhedral Projection. Ph.D. Thesis, Department of Combinatorics and Optimization, University of Waterloo.

[15] Martin, R. K. 1991. Using Separation Algorithms to Generate Mixed Integer Model Reformulations. *Operations Research Letters* **10**, 119-128.

[16] Muth, J. F., and G. L. Thompson. 1963. *Industrial Scheduling*, Prentice-Hall, Englewood Cliffs, NJ.

[17] Nemhauser, G. L. and L. A. Wolsey. 1988. *Integer and Combinatorial Optimization*. Wiley and Sons, New York.

[18] Rardin, R. L. and L. A. Wolsey. 1990. Valid Inequalities and Projecting the Multicommodity Extended Formulation for Uncapacitated Fixed Charge Network Flow Problems. Research Report CC-90-2, Institute for Interdisciplinary Engineering Studies, Purdue University.

[19] Taillard, E. 1994. Parallel Taboo Search Techniques for the Job Shop Scheduling Problem. *ORSA Journal on Computing* **6**, 108-117.

[20] van Laarhoven, P., E. Aarts, J. Lenstra. 1988. Job Shop Scheduling by Simulated Annealing. Report OS-R8809, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

A Constraint Satisfaction Approach to Makespan Scheduling

Cheng-Chung Cheng and Stephen F. Smith *
The Robotics Institute
Carnegie Mellon University
Pittsbur gh, PA 15213

February 1, 1995

1 Introduction

In this paper, we consider the application a constraint satisfaction problem solving (CSP) framework recently developed for deadline scheduling to more commonly studied problems of schedule optimization. Our hypothes is is two-fold: (1) that CSP scheduling techniques can provide a basis for developing high-performance approximate solution procedures in optimization contexts, and (2) that the representational assumptions underlying CSP models allow these procedures to naturally accommodate the idiosyncratic constraints that complicate most real-world applications. We focus specifically on the objective criterion of makespan minimization, which has rece ived the most attention within the job shop scheduling literature. We define an extended solution procedure somewhat unconventionally by ref ormulating the makespan problem as one of solving a series of different but related deadline scheduling problems, and embedding a simple CSP procedure as the subproblem solver. We summarize results of an empirical evaluation of our procedure performed on a range of previously studied benchmark problems. Our procedure is found to provide strong cost/performance, producing solution s competitive with those obtained using recently reported shifting bottleneck search procedures (Adams et al., 1988; Balas et al., 1993) at reduced computational expense. To demonstrate genera lity, we also consider application of our procedure to a more complicated, multi-product hoist scheduling problem(Yih, 1994). With

*This research has been sponsored in part by the National Aeronautics and Space Administration, under contract NCC 2-531, by the Advanced Research Projects Agency under contract F30602 -90-C-0119 and the CMU Robotics Institute.

only minor adjustments, our procedure is found to significantly outperform previously published procedures for solving this problem across a range of input assumptions.

We first introduce PCP (Smith and Cheng, 1993; Cheng and Smith, 1994), our previously developed procedure for deadline scheduling; then we define an extended procedure, called Multi-PCP, for makespan minimization; and finally we summarize the two computational studies.

2 Constraint Satisfaction Scheduling

Constraint satisfaction problem solving (CSP) has long been an area of active research within the field of Artificial Intelligence, and CSP models and heuristics have increasingly been investigated as a means for solving scheduling problems (Cheng and Smith, 1994; Minton et al., 1992; Muscettola, 1993; Sadeh, 1991; Smith and Cheng, 1993). Much of this work has focused on variations of the job shop deadline problem. A job shop deadline problem involves synchronization of the production of n jobs in a facility with m machines, where (1) each job j requires execution of a sequence of operations within a time interval specified by its ready time r_j and deadline d_j, and (2) each operation O_i requires exclusive use of a designated machine M for a specified amount of processing time p_i. The objective is to determine a schedule for production that satisfies all temporal and resource capacity constraints. The job shop deadline problem is known to be NP-Complete (Garey and Johnson, 1979).

There are different ways to formulate this problem as a CSP. Most frequently, it has been formulated as a problem of finding a consistent set of start times for each operation of each job. The PCP procedure of interest here, alternatively, is rooted in a problem representation akin to a disjunctive graph formulation (Balas, 1969). The problem is assumed to be one of establishing sequencing constraints between those operations contending for the same resource. We define a set of decision variables $Ordering_{ij}$ for each (O_i, O_j) such that $M_i = M_j$, which can take on two possible values: $O_i \prec O_j$ or $O_j \prec O_i$.

2.1 Problem Representation

The PCP scheduling model can be formalized more precisely as a type of *general temporal constraint network (GTCN)* (Meiri, 1991). In brief, a GTCN T consists of a set of variables $\{X_1, ..., X_n\}$ with continuous domains, and a set of unary or binary constraints. Each variable represents a specific temporal object, either a time point (e.g., a start time st_i or an end time et_i) or an interval (e.g. an operation O_i). A constraint C may be qualitative or metric.

A qualitative constraint C is repr esented by a disjunction $(X q_1 X_j) \vee ... \vee (X_i q_k X_j)$, alternatively expressed as a relation set $X_i \{q_1, ..., q_k\} X_j$, where q_i represents a *basic qualitative constraint.* Three types of basic qualitative constraints are allowed:

1. interval to interval constraints - The GTCN definition of (Meiri, 1991) includes Allen's 13 basic temporal relations (Allen, 1983): *before, after, meets, met-by, overlaps, overlapped-by, during, contains, starts, started-by, finishes, finished-by,* and *equal.* For convenience, we additionally include the relations *before-or-meets* and *after-or-met-by,* which represent the union of relation pairs (*before, meets*) and (*after, met-by*) respectively (Bell, 1989).

2. point to point cons traints - The relations identified in (Vilain and Kautz, 1986), denoted by the set $\{<, =, >\}$, are allowable here.

3. point to interval or interval to point constraints - In this case, the 10 relations defined in (Ladkin and Maddux, 1989) are specifi able, including *before, starts, during, finishe s, after,* and their inverses.

A metric con straint C is represented by a set of intervals $\{I_1, ..., I_k\} = \{[a_1, b_1], ..., [a_k, b_k]\}$. Two types of metric constraints are specifiable. A unary constraint C_i on point X_i restricts X_i's domain to a given set of intervals, i.e. $(X_i \in I_1) \vee ... \vee (X_i \in I_k)$. A binary constraint C_{ij} between points X_i and X_j restricts the feasible values for the distance $X_j - X_i$, i.e., $(X_j - X_i \in I_1) \vee ... \vee (X_j - X_i \in I_k)$. A special time point X_0 can be introduced to represent the "origin". Sin ce all times are relative to X_0, each unary constraint C_i can be treated as a binary constraint C_{0i}.

A GTCN forms a *directed constraint graph,* where nodes represent variables, and a edge $i \rightarrow j$ indicates that a constraint C_{ij} between variables X_i and X_j is specified. We say a tuple $X = (x_1, ..., x_n)$ is a *solution* if X satisfies all qualitative and metric constraints. A network is *consist ent* if there exists at least one solution. Figure 1 depicts the constraint graph for a simple 2 job, 2 machine deadline scheduling problem.

An enumerative scheme for solving a GTCN is given in (Meiri, 1991). Let a *labeling* of a general temporal constraint network, T, be a selection of a single disjunct (relation or interval) from each constraint specified in T. In the graph of Figure 1 there are 4 possible labelings, owing to the {*before-or-meets, after-or-met-by*} relation sets introduced to avoid resource contention between operation pairs (O_1, O_2) and (O_3, O_4). Sin ce any basic qualitative constraint can be trans lated into at most four metric constraints (Kautz and Ladkin, 1991) (e.g., O_i *before-or-meets* O_j translates to $et_i \leq st_j$), any labeling of T defin es a Simple Temporal Problem (STP) network - a metric network containing only single interval constraints (Dechter et al., 1991). T will be consistent if and only if ther e exists a labeling whose associated STP is consistent.

For any STP network, we can define a directed edge-weighted graph of time points, G_d, called a *distance graph.* An STP is consistent if and only if the corresponding

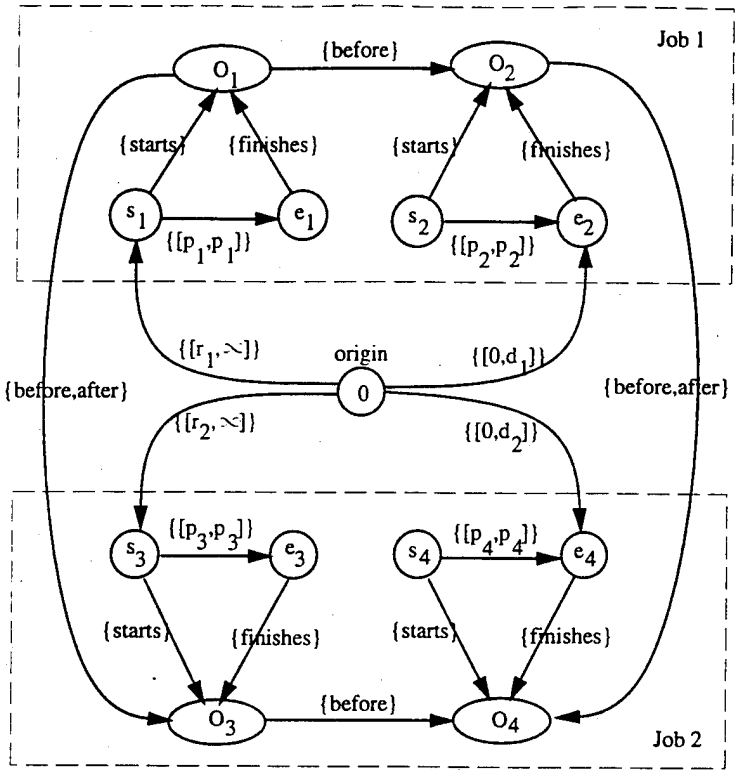


Figure 1: Constraint Graph for simple 2 job, 2 machine problem

distance-graph G_d has no negative weight cycl es. The *minimal network* of the STP can be specified by a complete directed graph, called the *d-graph*, where each edge, $i \rightarrow j$, is labeled by the shortest path length, sp_{ij}, from point i to point j in G_d (Dechter et al., 1991). An STP network can be solved in $O(n^3)$ time by the Floyd-Warshall's all-pairs short est-paths algorithm, where n is the number of variables in the STP network.

Thus, a simple, complete procedure for solving a GTCN is to enumerate all labelings, solve each corresponding STP and combine results. We can increase the efficiency of this enumeration procedure by running a backtracking search over a *meta- CSP* network, whose variables correspond to arcs in the GTCN that can be labeled in more than one way and whose domains are simply the set of possible labelings. In the case of the deadline scheduling problem, this leads to the set of decision variables $V = \{Ordering_{ij}\}$ previously identi fied, and a worst case complexity of $\mathcal{O}(n^3 2^{|V|})$.

2.2 The PCP Procedure

The PCP scheduling model (Smith and Cheng, 1993; Chen g and Smith, 1994) augments this basic backtracking search procedure to incorporate simple analysis of the temporal flexibility associated with each sequencing decision that must be made. This analysis

is utilized in two ways: (1) to specify dominance conditions that allow identi fication of unconditional decisions and early search space pruning, and (2) to provide heuristic guidance for variable and value ordering (i.e.. decisio ns as to what variable to assign next and what value to assign).

Specification and use of dominance conditions in PCP derives directly from the concept of *Constraint-Based Analysis* (CBA) origi nally developed in (Erschler et al., 1976; Erschler et al., 1980). This work utilized cal culations of the temporal slack associated with an unordered operation pair to distinguish among cases where neither ordering alternative, just one ordering alternative, or either alternative remains feasible. For example, if $slack(O_i \prec O_j) = lft_j - est_i - (p_i + p_j) < 0$ then O_i cannot be sequenced before O_j. These conditions are applied to detect and post any "forced" sequencing constraints at each step of the search, and to detect inconsistent solution sta tes.

In (Cheng and Smith, 1994), these dominance conditions are generalized to account for the wider range of constraints that are specifiable in a GTCN . Suppose $Ordering_{ij}$ is a currently unassigned variable in the meta-CSP network, and consider the d-graph associated with the current partial solution. Let s_i, e_i, s_j, and e_j be the sta rt and end points respectively of operatio ns Q and O_j, and further assume sp_{ij} is the shortest path length from e_i to s_j and sp_{ji} is the shortest path length from e_j to s_i. Then, four mutually exclusive cases can be identified:

Case 1. If $sp_{ij} \geq 0$ and $sp_{ji} < 0$, then $O_i \prec O_j$ must be selected.

Case 2. If $sp_{ji} \geq 0$ and $sp_{ij} < 0$, then $O_j \prec O_i$ must be selec ted.

Case 3. If $sp_{ji} < 0$ and $sp_{ij} < 0$, then the partial solution is inconsistent.

Case 4. If $sp_{ji} \geq 0$ and $sp_{ij} \geq 0$, then either ordering rela tion is still possible.

We note that the "slack-based" dominance conditions of (Erschler et al., 1976) represent a special case of the above conditions; under classical job shop scheduling assumptions (i.e., fixed processing times, simple job precedence constraints) $slack(O_i \prec O_j) = sp_{ij}$. However, many practical scheduling problems require satisfaction of more complex temporal constraints (e.g., bounded delays between job steps, minimum and maxi mum processing time constraints, inter-job synchroni zation). Under such more complex modeling assumptions, shortest path information provides stronger domi nance criteria.

The second distinguishing aspect of PCP is its use of sequencing flexibility analy sis for variable and value ordering, which dic tates how the search should proceed in the undecided states (case 4 above). Intuitively, in situations where several $Ordering_{ij}$ decisions remain to be made, each with both possibilities still open, we would like to focus attention on the decision that has the least amount of sequencing flexibility.

Conversely, in making the selected ordering decision, we intuitively prefer the ordering relation that leaves the search with the most degrees of freedom.

One very simple estimate of the sequencing flexibility associated with a given $Ordering_{ij}$ is the minimum shortest path length, $\omega_{ij} = min(sp_{ij}, sp_{ji})$, which gives rise to a variable ordering heuristic that selects the $Ordering_{ij}$ with the minimum ω_{ij}. This heuristic makes reas onable sense; at each step, the decision which is clo sest to becoming forced is taken. However, its exclusive reliance on ω_{ij} values can lead to problems. Consider two ordering decisions $Ordering_{ij}$ with associated shortest path lengths $sp_{ij} = 3$ and $sp_{ji} = 100$, and $Ordering_{kl}$ with $sp_{kl} = 4$ and $sp_{lk} = 4$. In this case, there are only limit ed possibilities for feasibly resolving $Ordering_{kl}$ and deferring this decision may well eliminate them. while a feasible assignment to $Ordering_{ij}$ is not real ly in any jeopardy.

To hedge against these situations, PCP instead bases variable ordering decis ions on a slightly more complex notion of *biased* shortest path length. Specifically, $bsp_{ij} = sp_{ij}/\sqrt{S}$ and $bsp_{ji} = sp_{ji}/\sqrt{S}$ are computed, where $S = \min\{sp_{ij}, sp_{ji}\}/\max\{sp_{ij}, sp_{ji}\}$ estimates the degree of similarity between the two values sp_{ij} and sp_{ji}. The sequencing flexibility associated with a given decision $Ordering_{ij}$ is redefined to be $\omega_{ij} = min(bsp_{ij}, bsp_{ji})$, and the decision selected during variable ordering is the dec i- sion with the minimum ω_{ij}. The value ordering heuristic utilized in PCP simply selects the ordering relation implied by $max(bsp_{ij}, bsp_{ji})$, i.e. the sequencing constraint that retains the most temporal flexib ility is posted.

2.3 More Efficient, Approximate Procedures

The dominance conditions and variable/value ordering heuristics that distinguish the basic PCP procedure do not, of course, change the exponential worst case behavior of the backtracking search required to guarantee comp leteness. Given our pragmatic interest in sol ving large problems, we thus introduce two less-costly, approximate solution procedures for later use. The first variant is simply de fined as a backtrack-free version of the basic PCP procedure. In particular, total reliance is placed on the ability of the search to move directly to a feasible solution; if Case 3 above is ever encountered (i.e., no feasible ordering for a given ordering decision), the search simply terminates in failure (and does not produce a solution). The effectiveness of this parti al solution procedure, which we will refer to as "Simple PCP" below, was demonstrated in (Smith and Cheng, 1993) on a set of previously publishe d CSP scheduling benchmark problems.

We also define a second variant, referred to below as "Simple PCP with Relaxation" which extends Simple PCP in the following manner. Whenever an orderi ng decision is recognized as Case 3, the unresolvable decision is set aside, and the search is allowed to proceed with other, still resolvable ordering decisions. Once all feasibly resolvable

decisions have been made, the set U of unresolvable (Case 3) decisions is then reconsidered. For each $Ordering_{ij}$ in U, deadlines d_i and d_j are relaxed (i ncreased) by $|max(sp_{ij}, sp_{ji})|$ and the corresponding precedence relation (which is now feasible) is posted. This second approximate procedur e thus always produces a solution. albeit one that may not satisfy all original problem constraints. Both approximate procedures can be seen to have worst case time complexity of $\mathcal{O}n^3|V|$, where $|V|$ is the number of ordering decisions that must be made.

3 MULTI-PCP

The applicability of constraint satisfaction schedul ing procedures such as PCP to more commonly stud ied problems of schedule optimization is not obvious. Here, we focus specifically on the problem of makespan minimization and propose one possible approach to incorporating these techniques. Our approach is motivated by the concept of problem duality exploited in the MULTIFIT algorithm (Coffman et al., 1978) in the context of multiprocessor scheduling. Suppose that we are given an instance of a makespan problem, denoted by $\Pi_M(I)$ where I represents the problem data associated with this problem instance. If we know the minimum makespan for $\Pi_M(I)$ to be C^*_{max}, then we can reduce $\Pi_M(I)$ to a special deadline problem $\Pi_D(I, d)$, where each job is assigned a 0 ready time and a common deadli ne d, with $d = C^*_{max}$. For any $d \geq C^*_{max}$, we are ass ured that a feasible solution to $\Pi_D(I, d)$ exists. More important, C^*_{max} defines a unique common deadline such that for $d < C^*_{max}$, $\Pi_D(I, d)$ has no feasible solut ion. This dual relationship between problems $\Pi_M(I)$ and $\Pi_D(I, d)$ implies that the makespan problem $\Pi_M(I)$ can be reformulated as a problem of finding the smallest common deadline, d_{min}, for which $\Pi_D(I, d)$ has a feasible solution.

Given an algorithm for optimally solving the deadline problem $\Pi_D(I, d)$, it is straightforward to construct an search procedure for determining d_{min} (and its associated schedule). We start with known upper and lower bounds d_U and d_L on the com mon deadline d_{min}; at each step, we attempt to solve $\Pi_D(I, d)$ for $d = (d_U + d_L)/2$. If a feasible schedule is found, d_U becomes d; otherwise, d_L becomes d. We continue the search until $d_U = d_L$, retaining the schedule with the best makespan as we go.

There is a complication, however, in utilizing this binary search procedure in conjunction with a heuristic deadline scheduling procedure. The search may fail to yield the best solution if the deadline scheduling procedure does not ensure *monotonicity* in solution results across an interval of common deadlines. This property implies tha t if a feasible solution cannot be found for a given common deadline d_1, then a solution will also not be found for any common deadline $d_2 < d_1$, and likewise if a solution is found for a given d_1, then a solution will also be found for any $d_2 > d_1$. It is not difficult to construct examples which demonst rate that neither of the simple, one-pass PCP procedures defined in Section 2.3 possess this property, and consequently the

assumptions underlying use of binary search are no longer valid. For this reason, we instead define our extended makespan minimization procedure in terms of a more conventional k-iteration search; the approximate PCP procedure (eit her variant) is applied k times with diff erent common deadlines evenly distributed between d_L and d_U. While k-iteration search obviously also provides no guarantee of finding the optimal solution, empirical analysis has indicated that, with proper selec tion of k, use of k-iteration search lead s to consistently better makespan minimization performance.

The only remaining issue concerns initial establishment of upper and lower bounds on d_{min}. A lower bound d_L is provided by the proced ure originally described in (Florian et al., 197 1), where each machine is sequenced independently in order of earliest operation start times and the maximum job completion time is then selected. An upper bound d_U can be obta ined through application of one or more priority dispatch rules (Panwalker and Iskander, 1977). In the experiments reported below, a set of six priority rules - SPT, LPT, LFT, EFT, MOR, and LOR - were applied, taking the best makespan generated as d_U. For all runs, the bound k on the number of iterations performed was set to 8.

4 Benchmark Problem Results

We applied two versions of Multi-PCP, defined by incorporating either Simple PCP or Simple PCP with Relaxat ion as the base CSP scheduling procedure, on two sets of previously studied benchmark problems. The first ("small") problem set consists of 39 job shop problems with sizes varying from 6-job by 6-machine to 15-job by 15-machine. The fir st three problems, Mt06, Mt10, and Mt20, are the long sta nding problems of (Fisher and Thompson, 1963). The remainder are taken from the 40 problems originall y created by (Lawrence, 1984); of these 40 probl ems, we include only the 36 problems for which optimal solutions have been obtained. The second ("large") set of benchmark problems, are the problems more recently defined by (Taillar d, 1993). This set consists of 80 larger job shop problems with sizes ranging from 15-job by 15-machine to 100-job by 20-machine. For each problem in this set, Taillard reported the "best solution" obtained with a tabu search proced ure that was run for extended time intervals.

We take as a principal comparative base, the shifting bottleneck family of procedures, SB1 , SB3 and SB4 (Adams et al., 1988; Balas et al., 1993), which provides a series of increasingly more accurate approximate procedures for makespan minimization at increasingly greater computational expense. We compare the performance of each of these procedures and Multi-PCP in terms of two meas ures: % deviation from the optimal solution (or best tabu search solution in the case of the large problem set) and amount of computation time required. Results for SB1 were obtained on a Sun SPARC 10 workstation using an implementation kindly provided to us by Applegate and Cook (for detail please see (Applegate and Cook, 1991)). Results for SB3 and

SB4 were taken from (Balas et al., 1993). with the reported Sun SPARC 330 computation times translated to reflect expected performance on a SPARC 10. Multi-PCP computation times were also obtained on a SPARC 10. All procedures considered were implemented in C. Sin ce SB3 and SB4 results have not been reported for the large problem set, comparison here is rest ricted to Multi-PCP and SB1.

Table 1 summarizes the performance results obtained on the small benchma rk problem set (with results aggregated according to problem size for the Lawrence problems). Associa ted computation times are given in Table 2. Compu tation times were found to be identical for both Multi-PCP configurations at the level of precisi on reported and are thus listed only once. [Detailed results for each individual problem for both problem sets are reported in (Cheng and Smith, 1995); space constraints prevent their inclusion here.]

First, we see that augmenting the base PCP procedure to produce "relaxed" deadline solutions when feasible solutions are not found yielde d improved solutions in only a small number of problems. In these isolated cases, however, the improvement provided by the exten ded procedure was sometimes substantial; for the mt20 problem of Fisher and Thompson, % deviation from the optimum was reduced from 8.76 to 2.32, matching the best solution found for this problem by any of the shifting bottleneck procedures. Since the extended Multi-PCP with Relaxation procedure incurs virtually no additional computational cost, we restrict attention to the results obtained with this configuration.

The makespan minimization perf ormance of Multi-PCP on the small problem set falls within the performance continuum defined by the shifting bottleneck procedures. On average, Multi-PCP is seen to perform better than SB1 and very close to SB3, with SB4 yielding the best overall makespan performance. Relative performance was found to vary across different problem subsets. On the three classic Fisher and Thompson problems, Multi-PCP found equivalent or better solutio ns than both SB1 and SB3 in all cases, and fai led to match the performance of SB4 in just one cas e. There is little difference in performanc e on the very small, 6-machine problems; all procedu res produce optimal or near optimal solutions in these problem categories. The results on the lar ger, 10-machine problem categories reveal perhaps the most significant comparative performance tren d. For problems with low ratios of number of jobs to number of machines, Multi-PC P exhibits its strongest comparative performance. In the case of the 10x10 problem category, Multi-PCP performed better on average than both SB1 and SB3, and very close to SB4. Conversely, Multi-PCP was found to be less effective (comparatively) on problems with high job to machine ratios. On the 30x10 problems (which turn out to be the easiest 10-machine problems for all procedures), all three shifting bottleneck procedures were able to obtain optimal solutions, wherea s Multi-PCP failed to find the optimum for 2 of the 5 problems in this category. Computat ionally, Multi-PCP's solution times on this problem set are seen to be comparable overall to those of SB1.

Table 3 extends the performance comparis on of Multi-PCP and SB1 to the larger problem set of Taillard. Corresponding average computation times by problem category are given in Tabl e 4. Ignoring the scalability problems encountered with the tested SB1 implementation (it couldn't solve some problems due to memory problems), the results at larger problem sizes make much more expl icit the comparative performance trends observed at the 10-machine problem level. Multi-PCP is seen to consistently outperform SB1 at low job-to -machine ratios, while the inverse is true at high job-to-machine ratios. Both procedures achieve increasingly better solutions at higher job-machine ratios (consistent with Taillard's observatio n that these problems are easier), but in no cases does either Multi-PCP or SB1 achieve the best solutions generated by extended Tabu search.

Examination of relative computational costs indicates some additional scalability trends and tradeoffs. Multi-PCP was found to consistently produce solution s in less computation time than SB1; the largest differential (roughly 4 times as fast) was observed in the problem categories with the smallest job-to-machine ratios, and, on average, Multi- PCP obtained solutions in about half as much CPU time as SB1. Multi-PCP was also found to be much more predictable with respect to computati onal cost. The variance in Multi-PCP solution times across all problem categories was extremely low in comparison to SB1.

5 The Hoist Scheduli ng Problem

The above study relates the performance of Multi-PCP to state-of-the-art makespan minimization procedures; perhaps somewhat surprising, it shows that Multi-PCP's use of a CSP schedul ing model in conjunction with fairly simple search control heuristics yields respectable performance (al though certainly not outperforming all previously reported results). A complementary consideration is its broader applicability to more idiosyncratic problem formulations.

To demonstrate generality, we consider application of Multi-PCP to a less-structured makespan minimization problem: the multi-product version of the hoist scheduling problem(Yih , 1994). The problem finds its origin in printed circuit board (PCB) electroplating facilities. In brief, a set J of jobs, $J = \{J_1, ..., J_n\}$ each require a sequence of chemical baths, which take place within a set M of m chemical tanks, $M = \{1, ..., m\}$. Execution of a particular chemical bath operation O_i req uires exclusive use of tank m. The processing time of any O_i required for a job j is not rigidly fixed; instead there is a designated minimum time, p_i^{min}, that j must stay in the tank for the bath to acco mplish its intended effect and a maximum time, p^{max}, over which product spoilage occurs. All jobs move through the chemical tanks in the same order, tho ugh a given job may require only a subset of the baths and thus "skip" processing in one or mor e tanks along the way. All job movement through the facility is accomplished via a

single material handling hoist, H, whi ch is capable of transporting a job initially into the system from the input buffer, from tank to tank, and finally out of the system into the output buffer. H can grip only a single job at a time, moves between any two adjacent stations (input buffer, tanks, or output buff er) at constant speed s, and has constant loading and unloading speeds, L and U, at any tank or buffer. The facility itself has no internal buffering capability; thus jobs must be moved directly from one tank to the next once they have entered the system. The objective is to maximize facility thro ughput (or equivalently minimize makespan) subject to these process and resource constraints.

Most previous work in hoist scheduling has consi dered simplified versions of this problem. The single-product, hoist scheduling problem has received the most attention (Phillips and Unger, 1976; Shapi ro and Nuttle, 1988; Lei and Wang. 1991; Armstrong et al., 1994). In (Yih and Thesen, 1991; Yih et al., 1993), a hoist scheduling problem involvin g a multi-product facility is considered, but without permitting variance in job routings (i.e. no tank skipping). To our best knowledge, only (Yih, 1994) has reported procedures for solving the general hoist scheduling problem defined above.

5.1 Extensions

The GTCN formalism introduced in Section 2.1 requires only slight extens ion to model the hoist scheduling problem. The only constraints that are not directly formulatable are those relating to synchronization of competing hoist (or material movement) operations; in this case, basic qualitative relations are insufficient, as they do not allow accounting of the "setup" time that may be required to position the hoist at the loading location. To overcome this limitation, we extend our representation of qualitative constraints to optionally include a metric quantifier. For purposes here, it is sufficient to include only the following two extended relations: *before-or-meets[lagtime]* and *after-or-met-by[lagtime]*, where $lagtime \geq 0$ designates a minimum metric separation between the related intervals. Thus, whereas the constraint O_i *before-or-meets* O_j implies $et_i \leq st_j$, the extended constraint O_i *before-or-meets*$[h_{ij}]$ O_j implies $et_i + h_{ij} \leq st_j$. For each pair of hoist operations O_i and O_j belonging to different jobs, we specify the constraint O_i {*before-or-meets*$[h_{ij}]$, *after-or-met-by*$[h_{ji}]$} O_j, where $h_{ij} = s * |destination_i - origin_j|$ and $h_{ji} = s * |destination_j - origin_i|$. [see (Cheng and Smith, 1995) for a complete discussion of how other aspects of the hoist problem are modeled and a simple example].

The presence of sequence-dependent setups also impacts the variable and value ordering heuristics utilized with the base PCP procedure. Recall from Section 2.2, that these heuristics rely on shor test path lengths as a basic indicator of sequencing flex ibility. In essence, the shortest path from et_i to st_j for operations O_i and O_j, designated sp_{ij}, indicates the current maximum feasible separation between these two points. However, shortest path lengths provide only a partial (distorted) view of maximum

separation if sequence-dependent setup delays are required. To sharpen the heuristics, we generalize the basic measure of flexibility in PCP to incorporate sequence-dependent lag times . Assume h_{ij} to be the lag time required if O_i is processed before O_j, and h_{ji} be the lag time required if O_j is processed before O_i (i.e., the constraint specified in the network is O_i {*before-or-meets*$[h_{ij}]$, *after-or-met-by*$[h_{ji}]$} O_j). We revise the dominance conditions and search control heuristics specified in Section 2.2 by simply substituting the extended calculation $(sp_{ij} - h_{ij})$ for sp_{ij} and, likewis e, substituting $(sp_{ji} - h_{ji})$ for sp_{ji}. Note that these revised definitions continue to accommodate the basic {*before, after*} relation (in which case, h_{ij} and h_{ji} are both set to the smallest possible temporal increme nt), as well as the basic {*before-or-meets, after-or-met-by*} relation set (where $h_{ij}, h_{ji} = 0$).

5.2 Results

To assess perfor mance, we carried out computational study following the same experimental design of (Yih, 1994). A PCB electroplating facility with 5 chemical tanks was assumed. All problems generated consisted of 100 jobs, each with randomly generated routings and tank processing time constraints, and all assumed to be simultaneously available. Since mater ial flow is uni-directional, differences in job routings correspond to which and how many tanks are skipped. Experiments were conducted to evaluate performance along two dimensions relating to facility constraints and operation: first as function of the relative speed of the hoist to mean tank processing time, and second as a function of the degree of flexibility provided by tank processing time constraints. To calibrate results, problems were also solved using the hoist scheduling procedure previously developed by Yih (Yih, 1994), designated below as the "Yih94 algorithm". Both procedures were implemented in C and run on a Sun SPARC 10 workstation.

In configuring Multi-PCP for these experiments, a simpler, "basic algorithm", used in (Yih, 1994) as a baseline for comparison, was incorporated to provide the upper bound d_U on the common deadline interval; d_L was obtained by computing the minimum total required processing time (including hoist operations) for each job and taking the maxi mum. To provide a more computationally competitive alternative to Yih's "real-time" procedure, a simple problem decomposition method(N. Hirabayashi and Nishiyama, 1994) was also employed; the input problem was partitioned into subproblems with equal numbers of jobs (10 for these experi ments) and solved independently by Multi-PC P, with the results then randomly combined to produce the overall solution - yielding overall solution times of about 100 seconds.

We present only the results obtained from one of the experiments performed, on problem sets designed to vary the ratio $\gamma = \hat{p}^{min}/s$, where $\hat{p}^{min}$ is the mean minimum processing time of tank operations and s is the speed of the hoist in moving between adjacent sys tem locations. Figure 2 summarizes the the performa nce of Multi-PCP and Yih94 in this experiment. Values plotted for each γ ratio rep resent the average %

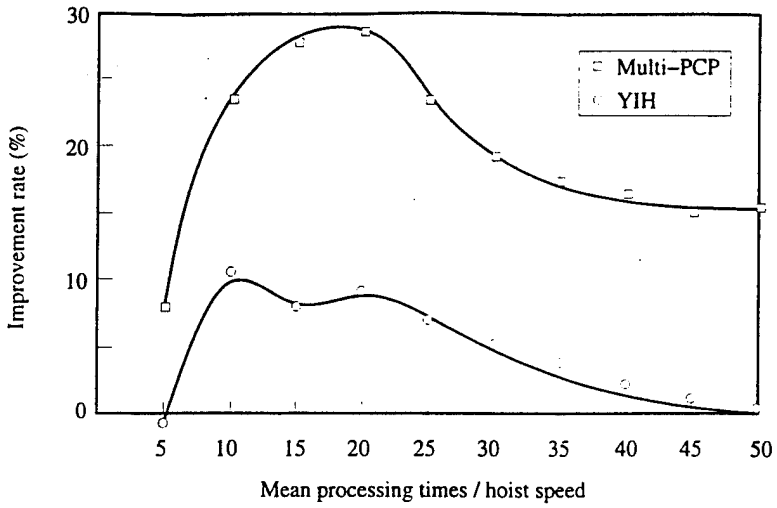


Figure 2: Solution improvement rates for increasing ratio of mean processing time to hoist speed

improvement over the basic algorithm on 10 randomly generated problems.

Both procedures are seen to generate the largest improvement for values of γ in the range of [10,25], with improvement rates degrading as γ becomes larger or smaller. In the case of Yih94, no improvement is obtained at either of the extreme points tested. Multi-PCP, alternatively, yields an improvement rate of 8% at the smallest γ value, and as γ becomes increasingly larger, its improvement rate stabilizes at about 15%. Across all experiments, Multi-PCP is seen to produce solutions that, on average, are 15% better (in relation to the baseline solution) than those obtained with Yih94.

Details of the full experimental design and all result s obtained are reported in (Cheng and Smith, 1995), and only strengthen the performance comparison.

6 Concluding Remarks

We have described a procedure for makespan scheduling based on formulation of the problem as a series of CSPs and iterative application of a CSP scheduling procedure with fairly simple search control heuri stics. It was shown to produce strong perform ance in relation to shifting bottleneck procedur es on benchmark scheduling problems. Perhaps more significant however, is the procedure 's generality. Real-world applications are often complicated by additional temporal synchronization and resource usage constraints, and solution procedures which rely on problem structure that is pecul iar to the canonical job shop problem formulation are of little use in such contexts. CSP sched uling models like Multi-PCP, alternatively, are based on very general representational assumptions and naturally extend to accommodate richer problem formulations.

References

Adams, J., Balas, E., and Zawack, D. (1988). The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391 - 401.

Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM*, 11(26):832–843.

Applegate, D. and Cook, W. (1991). A computational study of the job-shop scheduling problem. *ORSA Journal of Computing*, 3(2):149 - 156.

Armstrong, R., Lei, L., and Gu, S. (1994). A bounding scheme for deriving the minimal cycle time of a single-transporter n-stage processs with time window constraints. *European Journal of Operational Research*, 78:130–140.

Balas, E. (1969). Machine sequencing via disjunctive graphs: An implicit enumerated algorithm. *Operations Research*, 17:941–957.

Balas, E., Lenstra, J. K., and Vazacopoulos, A. (1993). The one machine problem with delayed precedence constraints and its use in job shop schedul ing. Technical report, Graduate School of Industrial Administration, Carnegie Mellon University, #MSRR-589(R).

Bell, C. (1989). Maintaining project networks in automated artificial intelligence planning. *Management Science*, 35(10):1192–1214.

Cheng, C. and Smith, S. (1995). Applying constraint satisfaction techniques to job shop scheduling. Technical Report CMU-RI-TR-95-03, The Robotics Institute, Carnegie Mellon University.

Cheng, C. and Smith, S. F. (1994). Generating feasible schedules under complex metric constraints. In *Proceedings of the Twelfth National Conference on Arti ficial Intelligence, Seattle, Washington.*

Coffman, E. G., Garey, M. R., and Johnson, D. S. (1978). An application of bin-packing to multip rocessor scheduling. *SIAM J. Comput.*. 7:1–17.

Dechter, R., Meiri, I., and Pearl, J. (1991). Temporal constraint networks. *Artificial Intelligence*, 49:61–95.

Erschler, J., Roubellat, F., and Vernhes, J. P. (1976). Findin g some essential characteristics of the fea sible solutions for a scheduling problem. *Operatio ns Research*, 24:772–782.

Erschler , J., Roubellat, F., and Vernhes, J. P. (1 980). Characterizing the set of feasible sequences for n jobs to be carried out on a single machine. *European Journal of Operational Research*, 4:189–194.

Fisher, H. and Thompson, G. L. (1963). Probabilistic learning combinations of local job- shop scheduling rules. In *Industrial Scheduling.* J. F. Muth, G. L. Thompson (eds), Prentice-Ha ll, Englewood Cliffs, NJ.

Florian, M., Trepa nt, P., and McMahon, G. B. (1971). An implicit enumeration algorithm for the machine sequencin g problem. *Management Science*, 17(12):B-782–B-792.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability. a Guide to the Theory of NP-Completeness.* W.H. Freeman Company.

Kautz, H. and Ladkin, P. B. (1 991). Integrating metric and qualitative temporal reasoning. In *Proceedings of the Nin th National Conference on Artificial Intelligence. Anaheim, CA.*, pages 241–246.

Ladkin, P. B. and Maddux, R. D. (1989). On binary constraint networks. Technical report, Kestrel Institute. Palo Alto, CA.

Lawrence, S. (1984). Resource constraint project scheduling: An experimental investigation of heuristic schedu ling techniques. Technical report, Graduate School of Industrial Administration, Carnegie-Mellon Uni versity.

Lei, L. and Wang, T. (1991). The minimum common-cycle algorithm for cyclic schedu l-ing of two hoists with time window const raints. *Management Science*, 37(12):1629–1639.

Meiri, I. (1991). Combining qualitative and quantitative constraints in temporal reasonin g. In *Proceedings of the Ninth National Confe rence on Artificial Intelligence, Anaheim, CA.*, pages 260–267.

Minton, S., Johnston, M. D., Philips, A. B., and Laird, P. (1992). Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205.

Muscetto la, N. (1993). Scheduling by iterative partit ion of bottleneck conflicts. In *Proceedings of the Ninth IEEE Conference on Arti ficial Intelligence Applications, Orlando, FL.*

N. Hirabayashi, H. N. and Nishiyama, N. (19 94). A decomposition scheduling method for operating flexible manufacturing systems. *International Jou rnal of Production Research*, 32(1):16 1–178.

Panwalker, S. S. and Iskander, W. (1977). A survey of scheduling rules. *Operations Research*, 25:45 - 61.

Phillips, L. and Unger, P. (1976). Mathematical programming solution of a hoist scheduling proble m. *AIIE Transactions*, 8(2):219–225.

Sadeh, N. (1991). Look-ahead techni ques for micro-opportunistic job shop scheduling. Technical report, CMU-CS-91-102, School of Computer Science, Carnegie Mellon University.

Shapiro , G. and Nuttle, H. (1988). Hoist scheduling for a pcb electroplating facility. *IIE Transactions*, 20(2):157–167.

Smith, F. S. and Cheng, C. (1993). Slack-based heuristics for constra int satisfaction scheduling. In *Proceedings of the Eleventh National Conference on Artificial Intelligence, Washington, DC.*, page s 139 – 144.

Taillard, E. (1993). Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64:278 – 285.

Vilain, M. and Kautz, H. (1986). Constra int propagation algorithms for temporal reasoning . In *Proceedings of the Fourth National Conference on Artificial Intelligence, Philade lphia, PA.*, pages 377–382.

Yih, Y. (1994). An algorithm for hoist scheduling problems. *Internat ional Journal of Production Research*, 32(3):501–516.

Yih, Y., Liang, T., and H.Moskowitz (1993). Robot scheduling in a circuit board production line. *IIE Transactions*, 25(2):26–33.

Yih, Y. and Thesen, A. (1991). Semi-markov decision models for real-time sched uling. *International Journal of Production Research*, 29(11):2331–2346.

Table 1: % deviation from optimal solution for Multi-PCP, SB1, SB3, and SB4 across small benchmark problem categories

Job x Machine	Multi-PCP		Multi-PCP w/ Relax		SB1		SB3		SB4	
	mean	σ	mean	σ	mean	σ	mean	σ	mean	σ
mt06	0.00	-	0.00	-	7.27	-	0.00	-	0.00	-
mt10	2.04	-	2.04	-	2.37	-	5.48	-	1.08	-
mt20	8.7 6	-	2.32	-	5.41	-	2.92	-	2.92	-
10 x 5	1.53	1.58	1.47	1.46	1.59	1.84	1.44	2.05	1.44	2.05
15 x 5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
20 x 5	0.00	0.0 0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
10 x 10	2.44	1.75	2.44	1.75	4.94	5.32	3.16	2.33	2.25	1.17
15 x 10	4.29	2.34	3.78	1.99	6.34	2.60	2.72	1.91	2.72	1.91
20 x 10	3.12	2.65	3.12	2.65	6.57	7.26	1.37	1.23	0.96	1.26
30 x 10	0.30	0.58	0.30	0.58	0.00	0.00	0.00	0.00	0.00	0.00
15 x 15	4.15	0.73	4.15	0.73	6.16	2.10	3.09	0.92	3.01	0.97
All	1.93	1.20	1.72	1.15	3.01	2.39	1.51	1.05	1.24	0.92

Table 2: CPU time (in seconds) for Multi-PCP, SB1, SB3, and SB4 across small benchmark problem categories

Job x Machine	Multi-PCP		SB1		SB3		SB4	
	mean	σ	mean	σ	mean	σ	mean	σ
mt06	0.05	-	0.12	-	0.78	-	1.45	-
mt10	0.38	-	0.72	-	2.21	-	7.76	-
mt20	1.38	-	0.28	-	1.99	-	3.62	-
10 x 5	0.13	0.09	0.12	0.01	0.40	0.20	0.56	0.28
15 x 5	0.22	0.26	0.15	0.02	0.12	0.04	0.12	0.04
20 x 5	0.07	0.07	0.16	0.03	0.14	0.04	0.14	0.04
10 x 10	0.26	0.03	0.67	0.13	1.61	0.16	3.20	0.25
15 x 10	1.04	0.10	1.20	0.18	3.41	0.98	5.74	2.52
20 x 10	2.51	0.13	1.61	0.26	3.86	2.27	7.50	5.35
30 x 10	4.85	3.96	2.58	0.58	4.13	1.76	4.13	1.76
15 x 15	1.29	0.17	4.55	1.30	13.34	0.49	26.79	1.05
Average	1.19	0.60	1.21	0.31	2.96	0.74	5.29	1.41

Table 3: % deviation from the best solution for Multi-PCP and SB1 across large benchmark problem categories

Job x Machine	Multi-PCP		Multi-PCP w/ Relax		SB1	
	mean	σ	mean	σ	mean	σ
15 x 15	5.74	0.74	5.57	0.78	9.00	2.05
20 x 15	7.52	1.82	7.27	1.49	10.15	2.14
30 x 15	9.88	2.57	9.65	2.47	8.38	3.06
50 x 15*	7.39	2.31	7.21	2.26	2.66†	1.57
20 x 20	7.60	1.54	7.33	1.33	9.98	2.29
30 x 20	11.76	2.43	11.64	2.37	13.05	2.55
50 x 20*	8.77	0.96	8.34	1.06	5.33†	1.82
100 x 20	4.88	1.4 1	4.88	1.41	- ‡	-
All*	8.38	1.72	8.14	1.65	8.36	2.21

† SB1 able to solve nine out of ten problems.
‡ SB1 unable to solve any of the 100x20 problems.
* Average performance is measured with respect to problems solved by both procedures.

Table 4: Mean and standard deviation of CPU seconds for procedures, Multi-PCP and SB1, performed on the large benchmark problems

Job x Machine	Multi-PCP		SB1	
	mean	σ	mean	σ
15 x 15	1.20	0.08	5.10	1.3 9
20 x 15	3.42	0.33	7.63	1.06
30 x 15	11.90	0.85	14.68	1.72
50 x 15	68.11	7.12	141.33	104.84
20 x 20	3.73	0.32	15.64	2.91
30 x 20	15.51	0.77	31.58	4.03
50 x 20	94.90	6.28	165.83	94.75
100 x 20	857.36	38.43	-	-

Combining the Large-Step Optimization with Tabu-Search: Application to The Job-Shop Scheduling Problem

Helena Ramalhinho Lourenco
Departamento de Estatística e Investigação Operacional
Faculdade de Ciências
Universidade de Lisboa
Campo Grande C/2
1700 Lisboa, Portugal
e-mail: helena@gist.deio.fc.ul.pt

Michiel Zwijnenburg
Department pf Econometrics
Faculty of Economy
Erasmus University
Rotterdam, The Netherlands

1. Introduction

We apply the combined technique of tabu-search and large-step optimization to the job-shop scheduling problem. The job-shop scheduling problem can be defined as follows: given a set of machines and a set of jobs, the objective is to construct a schedule which minimizes the time necessary to complete all the jobs.

In Section 2, we review local optimization methods, as local improvement and simulated annealing, and a two-phase optimization method, known as large-step optimization, which has recently been introduced for the traveling salesman problem. The first phase of this new method consists of a large optimized transition in the current solution, while the second phase is basically a local search method. The main advantage of the large-step optimization methods is the use of optimization and tailored procedures in combination with local search methods.

So far, the methods used in this second phase, also called small-steps phase, are the local improvement and the simulated annealing methods. In this work we combine the large-step optimization with a tabu-search approach and apply it to the job-shop scheduling problem, because both methods separately applied to the problem gave good results. We present this combined method and related computational results in Section 3.

In Section 4, we present some diversification strategies and relate them with the large-step optimization method. Relevant computational results are also presented.
In Section 5, we present some conclusions and possible future research ideas.

2. Local and large-step optimization methods

Local improvement methods are iterative methods where initially a feasible solution of the problem in question is obtained and at each step we make one local modification, or transition, of a prespecified type in the current solution. If an improvement in the cost function is obtained, then we accept the new solution as the current one, and otherwise we reject it. The algorithm terminates when we cannot improve the current solution by performing one transition, i.e., when we have a local optimal solution.

Consider now the job-shop scheduling problem. A solution corresponds to a schedule and the cost function is the maximum completion time of the schedule. The method used to obtain the initial schedule is the priority rule with priority function most work remaining. An extensive study considering different methods to obtain initial schedule has been done by Lourenco [1994].

Simulated annealing accepts solutions with increased value for the cost function, called uphill moves, with small and decreasing probability, in an attempt to get away from a local optimal solution and keep exploring the region of the feasible solutions. The probability is controlled by a parameter known as the temperature, which is gradually reduced from a high value, at which most uphill moves are accepted, to a low one at which few, if any, such moves are accepted.

In our implementation of simulated annealing, we use the same method to generate the initial schedule as for the local improvement method, and the neighborhood structure of van Laarhoven, Aarts and Lenstra [1992] (VLA&L), but we use a different cooling schedule. Our cooling schedule is very similar to the geometric cooling schedule presented in Johnson et al.[1989].

Martin, Otto and Felten [1992] introduced a large-step optimization method for the traveling salesman problem, which was applied to the job-shop scheduling problem by Lourenco [1993]. The large-step optimization method consists of three main routines: a large-step, a small-steps method and an accept/reject test, which all three are consecutively executed for a certain number of large-step iterations.

Local improvement methods proceed downhill for a while, making good progress, but they tend to get trapped in local optimal solutions, usually far away from the global optimal solution. Simulated annealing methods try to improve this by accepting uphill moves depending on a decreasing probability controlled by the temperature parameter. But, at small temperatures, they also tend to get stuck in valleys of the cost function. Large-step optimization methods allow to leave these valleys even at small temperatures, and only then a local search optimization method is applied, returning to a local optimal solution. At this point, an accept/reject test is performed.

The three main routines of a large-step optimization method are: the method to perform a large step, the one for the small steps and the accept/reject test.

The accept/reject test can accept only downhill moves. This looks like the local improvement methods presented before, but where only local optimal solutions (with respect to the small steps) are considered. On the other hand, the accept/reject test can accept all moves or it can also look like the test done in a simulated annealing method, i.e. all downhill moves are accepted and a uphill move is accepted with a small and decreasing probability.

The large step should be constructed so that valleys are easily climbed over, and they should be specially tailored to the problem in consideration. An important part of the large-step optimization methods is the large-step itself. As mentioned, this procedure should make a large modification in the current solution to drive the search to a new region and, at the same time, to perform some kind of optimization to obtain a solution not too far away from alocal (or global) optimum schedule. These two factors of the large-stepmethod are the main differences from local search and local search methods.

So far, the methods that are usually associated with the small steps are local optimization algorithms like the local improvement method or the simulated annealing method. In the next section, we will consider the tabu-search approach as the small-steps procedure.

Different applications of the large-step optimization methods have appeared in the literature. Martin, Otto and Felten [1992] developed a large-step optimization methods for the traveling salesman problem, where they used a 4-opt move as the large step, and, as the small steps, a local improvement method based on 3-opt moves. Johnson [1990] also applies a large-step local optimization method, that he called iterated Lin-Kernighan, to the traveling salesman problem, using a 4-opt move as the large-step and the Lin and Kernighan heuristic for the small steps. Lucks [1992] applied large-step optimization to

the graph partitioning problem, where the large step is done by interchanging a randomly selected set of a given size and, as small steps, the Kernighan and Lin approximation method and a simulated annealing approach were used. He concluded that for random graphs, large-step optimization outperformed all other tested algorithms, while for geometric graphs, the large-step method in some cases was outperformed.

The large step optimization is different from random restarts methods, Feo and Resende [1994], in which randomly constructing an initial schedule followed by a local optimization method is repeated for a number of iterations, returning the best solution found. As suggested by Johnson [1993], large-step optimization methods can be viewed as restart methods. But instead of starting with a different initial random solution at each iteration, in the large step a solution S' is obtained from a solution S, by performing a sufficient large optimized perturbation. In this way, all the achievements from previous runs are not totally lost in the next runs. After every large step a local optimization method is applied as a small steps procedure which leads the search process to a good local solution. Also, these methods have the advantage of combining the ideas associated with local search methods and tailored heuristics.

Next, we apply the large-step optimization methods to the job-shop scheduling problem. For the small steps, we consider the following methods: the local improvement method and the simulated annealing method.Lourenco [1994] proposed four methods to perform the large step procedure. The first three methods are based on choosing one or two machines and reordering the operations to be processed on these machines, using some prespecified method, and the last one is based on reversing the order of processing of several operations. We use one of the most successful which can be described as follows: reschedules sequencely two randomly chosen machines to optimality. First choose two random machines and ignore the order of the operations to be processed in these machines. Next for one of the two machines, determine the release and delivery times of the operations to be processed by the respective machine, given the scheduled operations on the other machines. Solve the one-machine schedule associated with this machine using Carlier's algorithm, Carlier [1982]. Then repeat this process to reschedule the second machine. For more information, see Lourenco [1994].

We will present computational results obtained from the application of all these methods to several instances of the problem. The methods are tested on set of known instances. Since the amount of testing is extensive, in preliminary tests we use just a subset of the instance. From the computational results we can conclude that the local improvement method is clearly inferior, even when several trials are performed. The large-step optimization methods outperformed the simulated annealing method, but the difference between these two became closer when the same amount of time was allowed for both. In many cases, the large-step optimization methods found an optimal schedule and in others the distance from the optimum or lower bound was small.

Given the results obtained by previous applications of the large-step optimization methods to combinatorial optimization methods, including the job-shop scheduling problem, and given the success of tabu-search to the same problem, it looks promising to apply the large-step optimization method with as small steps method the tabu-search to the job-shop scheduling problem.

3. Combined techniques of large-step optimization and tabu-search

The tabu-search, in contrary of local optimization methods refer previously, makes use of historical information. The historical information is kept in three kinds of memory functions: the short-term, the intermediate and long term memory function. The last two will be considered in next section. Short term memory function has the task to memorize certain attributes of the search process of the recent past and it is incorporated in the search via one or more tabu list. For more information on tabu-search methods see for

example Glover [1989] and Glover et al.[1993].

The application of tabu search to the job-shop scheduling problem by Dell'Amico and Trubian [1992] gave very good results, in small amounts of time. Note that a big percentage of the running time for a large-step optimization method is spent by the simulated annealing method. Therefore, an improvement for large-step optimization methods should be achieved by using small-step methods that give good local optimal solutions and run in a smaller amount of time than the simulated annealing. By the above observations the tabu search methods looks like a good candidate to be used instead of the local improvement method, that are fast but gives poor local optimal solutions, and instead of simulated annealing that gives good local optimal solutions, but at very high computational cost.

The basis for our implementation of the tabu-search is derived from Dell'Amico and Trubian [1993], but there are some differences. Dell'Amico and Trubian used a bi-directional method to obtain an initial schedule, while our tabu-search, like in the local improvement method and simulated annealing approach, starts by constructing a initial schedule according to the priority rule most remaining work. Due the objective of using the tabu-search method in the large-step optimization method, in our implementation we did not apply their intensification strategy which let the process return to the best solution found so far in the process, if during a certain number of iterations the best solution did not improve. Also different in our implementation is the use of exact methods for calculating the longest path in a graph belonging to a feasible solution and for checking if a graph contains a cycle. Similar to Dell'Amico and Trubian tabu-search are the neighborhood structure (D&T), the tabu-list structure and the rules which define the length of the tabu search.

The parameters in all programs are set in such a way that for a certain instance, they use the same amount of computer time as our implementation of the Dell'Amico and Trubian tabu-search method needed for 1200 iterations for the same instance.

We can conclude that the large-step optimization combined with the tabu-search outperforms the simulated annealing and the large-step optimization using simulated annealing. The average value out of 5 runs for the large-step optimization with tabu-search is always smaller than the average results for the large-step optimization with simulated annealing. In the beginning of the search in the simulated annealing approach there is no fast improve as in the tabu-search. Therefore the simulated annealing waists time in the beginning of the process, while this initial period is effectively used by tabu-serach. Even the tabu-search alone outperforms the large-step optimization method with simulated annealing small-steps procedure but it is slightly inferior to the large-step optimization with tabu-search. For example, for the famous instance of Muth and Thompson with 10 jobs and 10 machines (MT10), the best value obtained by the large-step optimization method with tabu search (LSTS) was the optimal value of 930 and the average was 939 meanwhile if the simulated annealing (LSSA) was used as the small-steps procedure the respective values were 951 and 963. Using our implementation of the tabu-search (TS) alone we obtained the following values, 930 was the best obtained and 945 the average. For the open instance propose by Lawrence (LA21) the best value obtained by LSTS was 1047, the best value that we are aware of, and the average was 1055. While for the LSSA the respective values were 1078 and 1096. Applying the TS method alone we obtained the following respective values 1059 and 1065.

We have also compared several large-step optimization methods, which differ from each other by the number of the large-step iterations. The combination of large/small steps number of iterations are set in a special proportion to each other such that the complete run of the algorithms takes as much time as one run of the previous large-step optimization method with tabu-search, 12000 total iterations. In general the best results in terms of the best and the average value were obtained by large step optimization with the number of large step iterations between 5 and 20, with a little drop near 10 large set

iterations.

Viewing the results for both tabu-search with different neighborhoods, it seems that the simple neighborhood structure (VLA&L) with a large number of iterations can compete with amore complicated neighborhood structure (D&T), which need more time to pick the best move out of all possible ones, and therefore may be executed for fewer iterations. But more tests in this issue is need to be able to make stronger conclusions.

4. Diversification strategies

Long term memory is the basis for the diversifying strategy which has to lead the process to new regions of the solution space. The objective of a diversification strategy is to produce a new starting point, wherefore the local search can continue.

In this section we pay attention to a number of diversification strategies, which are closely related to the large-step methods. Both a diversification strategy as well as a large-step method apply a big change to the current solution, which provides them the possibility to get the search process out of a local optima valley. The difference between both methods is that diversification strategies always make use of a long term memory, where, large-step optimization can apply a relevant big change to a given solution without the use of information directly from the past, but using optimization techniques.

In the approach to obtain a new starting point, two kind of diversifying strategies can be distinguished, strategies using a restart method and strategies using a method which lead the search to new regions by an iterative process. These last kinds of diversification methods usually provide advantages over restart methods, Glover[1993]. For a number of iterations the random restart methods construct a random initial schedule, followed by the application of a local optimization procedure, returning the best solution found. The only difference with a more systematic restart method is that the initial solution is not randomly obtained, but constructed with information from previous periods in the search process. The storage of this information can be done with frequency counts. The idea is to count the number of times that moves are applied during the non-diversification phase of the search process, while during the diversification phase the moves are penalized according to their frequency counts, with the objective to oblige the process to search for moves not so often applied, resulting in never visited solutions.

Next, we present two diversification methods for the job-shop scheduling problem using frequency counts. The first method is a restart method which begins every diversification iteration by constructing a new starting schedule using frequency counts obtained in the preceding steps of the method, that we will call small steps in analogy with the large-step optimization methods. Every time the method finds a better solution in the small steps phase, best solution and best value are updated and the frequency memorizes the predecessor and successor of every job on every machine in the new best schedule. After these period of small-steps is finished, the frequency counts for every job how many times a job was the predecessor and the successor of any another job at any machine in the improving schedules. We obtain a diversified schedule by constructing a schedule according to a priority rule, which can be overruled if it wants to schedule an operation after another operation resulting in an already existing neighbor combination in one of the memorized improving schedules. In this way, we try to obtain a schedule wherein the predecessor and successor of every job are different from the predecessors and successors of the same job in the same machine in the memorized improving schedules.

The second diversification we have implemented is an iterative method, using frequency counts, which leads on a path to new regions of the solution space. The frequency counts are obtained in the previous iterations by storing the number of times a certain arc has been reversed. In our implementation we memorized only the critical arcs, involved in a move. This is obvious for the VLA&L neighborhood. For the D&T neighborhood

structure, counting the exact number of reversals would be very time consuming if we consider all single reverses. Therefore, we decided to count only the number of time the critical arcs are involved in a move. The diversification method consist of running the same tabu-search method, but now the moves are penalized to the critical arcs involved in the following way (see Laguna [1992]): compare-length(S´)=length(S´)+ penalty*frequency-of-critical-arc, where S´ is the neighbor of S. If a critical arc of S is involved in the applied move, the number of times this arc was picked in the preceding steps is stored in the frequency-of-critical-arc. The penalty is the value which a move is punished per applied move involving the arc. In this way moves which were frequently applied in the preceding iterations are punished in the diversification phase to force the method to apply new moves with the objective to drive the search process to new regions.The value of the variable penalty and the number of iterations in the diversification phase will be determined via trial and error method.

We obtained results with this method for four combinations of number of diversification steps/small steps iterations. These combinations are similar to the ones used in testing the large-step optimization methods with tabu-search. With some exceptions the incorporation of our restart diversification strategy does not contribute to a better performance of the tabu-search method. We have observed that after the diversification iterations, a resulting schedule has a high length. In general, the best values, as well the average ones, are getting worse when the number of diversification steps increases. Therefore we can conclude that our restart diversification strategy changes the schedule too drastically. After the application of the diversification method, the tabu-search method needs a lot of iterations to get near to good solutions. For the method with 5 diversification step iterations, the good solutions are already difficult to reach, as its results in general are not better than the results of tabu-search without diversification.

Consider now the second diversification strategy, designated by iterative diversification strategy. When we compare the results obtained by this method (TSID) with tabu-search without diversification (TS) and large-step optimization with tabu-search (LSTS), we can conclude that on average this method can compete with LSTS and performs better than TS. As for example, for MT10 the best and average value obtained by TSID were 936 and 938, meanwhile for TS was 940 and 946, and for LSTS were 930 and 936. For LA21, the results were, for TSID and LSTS were 1047 and 1055, respectively.

The reason that the iterative diversification strategy performs better than the restart one is that, the iterative diversification with the chosen parameters changes the schedule enough to get out of an eventually bad valley, but not so radical that the resulting schedule has high length. The same happen when we apply the large-step procedure in the large-step optimization methods. In the iterative diversification strategy a kind of "punished optimization" is done, and with the right parameters, it does not take a lot of small-steps iterations to get near a good solution again, contrary to what happens in the restart diversification strategy.

5. Conclusions

The idea of combining large-step optimization method with a tabu-search approach seems interesting to solve combinatorial optimization problem, due the good results that we obtain from the application to the job-shop scheduling problem. Therefore, it will be interesting to develop similar methods to solve other problems and test the efficiency of these combined optimization methods. The combination of optimization and tailored methods with local search methods may lead to an improvement in the use of these last methods in solving problem. In our opinion, it is an very interesting area to develop.

Another issue that it will be relevant to explore is the connection between the large-step procedure in the large-step optimization methods and the diversification strategies, briefly mention in this work. The relation with the intensification strategies was not considered

in the work, but we are planning to do it in near future.

References

Carlier, J. [1982], "The One-Machine Sequencing Problem", European Journal of Operational Research, 11:42-47.

Dell'Amico, M. and M. Trubian, [1993], "Applying Tabu-Search to the Job-Shop Scheduling Problem", Annals of Operations Research, 41: 231-252.

Feo, T.A. and M.G.C. Resende, "Greedy Adaptive Search Procedures", to appear in Annals of Operations Research.

Glover, F. [1989], "Tabu Search - Part I", ORSA Journal on Computing, 1(3):190-206.

Glover, F. [1993], "Tabu Thresholding: Improved Search in Nonmonotonic Trajectories", to appear in ORSA Journal on Computing.

Glover, F, M. Laguna, T. Taillard and D. de Werra [1993], "Tabu Search", Annals of Operations Research, vol.41, no. 1-4.

Johnson, D.S. [1990], " Local Optimization and the Traveling Salesman Problem", Proceedings of the 17th Annual Colloquim on Automata, Languages and Programming, Springer-Verlag, 446-461.

Johnson, D.S. [1993], "Random starts for local optimization", DIMACS Workshop on Randomized Algorithms for Combinatorial Optimization.

Johnson, D.S., C.R. Aragon, L.A. McGeoch, and C. Schevon, "Optimization by Simulated Annealing: an experimental evaluation; part I, graph partitioning", Operations Research, 39(3):865-892.

Laguna, M. [1992], "Tabu-search primer", preprint.

Lourenco, H.R. [1994], "Job-Shop Scheduling: Computational Study of Local Search and Large-Step Optimization Methods", to appear in European Journal of Operational Research.

Lucks, V.B.F. [1992], "Large-Step Local Improvement Optimization for the Graph Partitioning Problem", Master's Dissertation, Cornell University, Ithaca, NY, USA.

Martin, O., S.W. Otto and E.W. Felten [1992], "Large-Step Markov Chains for the TSP Incorporating Local Search Heuristics", Operations Research Letters, 11: 219-224.

van Laarhoven, P.J.M., E.H.L. Aarts and J.K. Lenstra, "Job Shop Scheduling by Simulated Annealing", Operations Research, 40(1):113-125.

An Overview of Learning in the Multi-TAC System

Steve Minton John A. Allen* Shawn Wolfe
Andrew Philpot

Recom Technologies
NASA Ames Research Center, M.S. 269-2
Moffett Field, CA 94035-1000
{minton,allen,shawn,philpot}@ptolemy.arc.nasa.gov

January 31, 1995

1 Introduction

MULTI-TAC is a system that synthesizes programs for solving combinatorial problems. It makes use of machine learning techniques to tailor its programs to particular distributions of instances. In discussing MULTI-TAC, we adopt the standard terminology of computer science and use the term "problem" to refer to a generic problem class and "instance" to refer to a particular problem instance. For example, "Graph-3-Colorability" [4] is a problem that requires that each node in a graph be assigned one of three possible colors such that no two neighbors have the same color. An instance of Graph-3-Colorability consists of a specific graph and a specific set of colors.

To generate a program, MULTI-TAC accepts a problem specification from the user as well as a set of sample instances or an instance generator. MULTI-TAC uses this information both to specialize a library of generic search algorithms and generic search heuristics, and to find the best combination of algorithm and search heuristics for solving the instances provided. The result is a LISP program with specialized data structures, constraint representations, and search control knowledge.

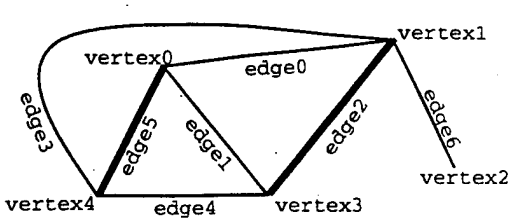
In this paper we give an overview of the MULTI-TAC system focusing on its learning mechanisms. We present one of the areas of current research in the project, and summarize some empirical results comparing the programs written by MULTI-TAC to those written by humans.

2 The Multi-TAC Architecture

In order to present a problem to MULTI-TAC, it must be formalized as an integer Constraint Satisfaction Problem (CSP), that is, as a set of constraints over a set of integer variables. A solution is a complete assignment of values to variables such that the constraints on each variable are satisfied.

For example, consider the NP-complete problem, "Minimum Maximal Matching" (MMM), described in [4]. An instance of MMM consists of a graph $G = (V, E)$ and an integer $K \leq \mid E \mid$. The problem is to determine whether there is a subset $E' \subseteq E$ with $\mid E' \mid \leq K$

*Also affiliated with the University of California, Irvine




```
(declare-parameter 'K 2)
(declare-type-size 'edge 7)
(declare-type-size 'vertex 5)
(declare-relation-data
  '((endpoint edge0 vertex0)
    (endpoint edge0 vertex1)
    (endpoint edge1 vertex0)
    (endpoint edge1 vertex3)...))
```

Figure 1: An instance of MMM with $K = 2$. A solution $E' = \{\text{edge2 edge5}\}$ is indicated in boldface. The instance specification is on the right.

such that no two edges in E' share an endpoint, and every edge in $E - E'$ shares an endpoint with some edge in E'. See Figure 1 for an example.

To formulate MMM as a CSP, we represent each edge in the graph with a variable. If an edge is chosen to be in E', it is assigned the value 1, otherwise it is assigned the value 0. The constraints can be stated as follows:

1. If $edge_i$ is assigned 1, then every $edge_j$ that shares an endpoint with $edge_i$ must be assigned 0.
2. If $edge_i$ is assigned 0, then there must exist an $edge_j$ such that $edge_i$ and $edge_j$ share an endpoint, and $edge_j$ is assigned 1.
3. The number of edges assigned 1 must be less than or equal to K.

A *problem specification* describes the types (e.g., *vertex* and *edge*) and relations (e.g., *endpoint*) and specifies the constraints in a typed predicate logic. An *instance specification* (Figure 1) instantiates the types and relations referred to in the problem specification. Our constraint language is relatively expressive, as it allows for full first-order quantification and the formation of sets and bags. Below we show how the first constraint above is specified, for some edge $Edge_i$:

```
(or (not (assigned Edge_i 1))
    (∀ Vrtx : (endpoint Edge_i Vrtx)
      (∀ Edge_j : (endpoint Edge_j Vrtx)
        (or (equal Edge_j Edge_i)
            (assigned Edge_j 0))))
```

The notation ($\forall x$: (endpoint x y)...) should be read as "for all x such that (endpoint x y)...".

The constraint language includes two types of relations: problem-specific *user-defined relations* such as *endpoint*, and built-in *system-defined relations*, such as *assigned*, *equal* and *less-than*. (There are no functions; instead, we use two-place relations.) The *assigned* relation has special significance since it represents the state during the search process. In MMM, for example, the search proceeds by assigning each edge a value. In a solution state, every edge must be assigned a value such that the constraints are satisfied.

The program synthesis process is hierarchically organized. At the top level, the system employs one of a set of generic constraint satisfaction search algorithms, including backtracking and iterative repair[8]. For the purposes of this paper, we refer only to the backtracking search. The backtracking algorithm operates by successively selecting a variable and then assigning to it a value. Backtracking occurs when all values for a variable fail to satisfy the constraints. Thus, two obvious points for search control knowledge are the choice of which variable to instantiate next and the choice of which value to assign. Associated with the backtracking schema are generic heuristics for choosing variables and values, including:

- Most-Constrained-Variable-First: This variable ordering heuristic prefers the variable with the fewest possible values left.
- Most-Constraining-Variable-First: A related variable ordering heuristic, this prefers variables that constrain the most other variables.
- Least-Constraining-Value-First: A value ordering heuristic, this heuristic prefers values that constrain the fewest other variables.

At the next level of program synthesis, MULTI-TAC first executes the first phase of its learning mechanism by operationalizing the generic variable and value-ordering heuristics, producing a set of candidate search control rules. This set may be large (typically between 10 and 100 rules in our experiments), since there are a variety of heuristics and each may be specialized and/or approximated in several different ways.

The second phase of MULTI-TAC's learning process determines (among other things) which subset of the generated search control rules should be incorporated into the final program. The system does this by hill-climbing through the space of *system configurations*, each of which consists of a generic search template, a combination of search control rules, and a variety of flag settings controlling other heuristic mechanisms (e.g., whether or not to use forward checking). The hill-climbing search is based on the *utility evaluation* of a given configuration. The evaluation is done by compiling a LISP program that implements the configuration, and then "experimenting" with the program by running it on a set of instances. (The compilation process also includes a variety of additional optimization techniques, such a finite differencing[13] and constraint simplification, which we will not discuss here).

3 Generating Heuristics

MULTI-TAC uses two different methods for generating problem-specific search control rules: one analytic, and one inductive.

3.1 An Analytic Approach

MULTI-TAC's analytic method of generating search control rules is derived from the work done by the Explanation Based Learning (EBL) community [6, 12, 3]. In essence, the process uses the problem specification to partially evaluate a generic heuristic. The partial evaluation is guided by a meta-theory which specifies how terms in the generic heuristic relate to terms in the problem specification.

For example, the generic heuristic for Least-Constraining-Value-First recommends that the problem solver choose the value that least constrains the value choices of other variables. One way of approximating this idea is to find the value that affects the fewest number of other variables. Another approximation is to find the value that knocks out the fewest number of possible values from other variables. Both of these approximations correspond to meta-theories in MULTI-TAC.

The analytic method builds search control rules by partially evaluating each of the generic heuristic using the problem specification and each of the meta-theories. For example, one of these partial evaluations uses the Least-Constraining-Value-First heuristic, the meta-theory "find the value that effects the fewest number of other variables" and the Graph-3-Colorability constraint, which states that a variable assigned $color_i$ cannot have any neighbors that are assigned $color_i$. The result is a specialized version of Least-Constraining-Value-First that says "choose the color that is in the domain of the fewest number of uncolored neighbors."

One of the advantages of this approach is that it allows one to generate several different approximations or specializations of a generic heuristic based on the number of different meta-theories. For instance, two different ways of approximating the Most-Constrained-Variable-First heuristic include:

- Choose the node with the smallest domain.
- Choose the node with the most uncolored neighbors.

Individually, these rules only produce minor reductions in search. However, MULTI-TAC is capable of composing search control rules such that the second rule can be used to break ties if the first rule does not produce a unique choice. When composed, these two rules produce the Brelaz[2] method, a well known technique for solving Graph-3-Colorability problems.

A second advantage of the analytic method is that the meta-theories need only know about the system defined types and relations, such as *variable*, *value* and *assigned*. The problem definition provides the mapping between the system defined relations and the user defined relations, such as *neighbor* and *color*, allowing the process to be used on any specifiable problem.

3.2 An Inductive Approach

MULTI-TAC's inductive method of generating search control rules is based on a form of generate and test. The generation phase uses a brute-force enumeration method to systematically construct all combinations of primitives, connectives and quantifiers in the problem specification language. The enumeration produces all well-formed formula starting from length one to length K, where length is defined as the number of component atomic expressions. The final phase of the generation process constructs search control rules using the formulas as antecedents.

The rules are tested on sample data to determine which of them might be useful. For example, consider how a candidate control rules approximating the Most-Constrained-Variable-First heuristic can be learned. The inductive method generates all candidate variable-ordering rules up to size K, and tests them using examples that illustrate the most-constrained heuristic. To find examples, we run our CSP problem solver (without any ordering heuristics) on randomly selected problem instances and periodically stop the solver at variable selection choice points. Each example consists of a pair of variables and a state, such that one variable is a most-constrained variable in that state and the other variable is not. A variable is "most-constrained" if no other variable has fewer possible values. We test each rule on each example by seeing if the antecedent holds for the most-constrained variable and does not hold for the other variable. In this case we say the rule was correct on the example. A sample set of rules generated by this process for Graph-3-Colorability is the following:

- Choose the node with the smallest domain.
- Choose a node adjacent to the node with the most uncolored neighbors.

This initial filtering removes the rules that are not worth investigating. However, since we do not expect our rules to be one-hundred percent correct, we retain all rules which are correct more often than they are incorrect. The second phase of learning is then used to find the best combination of these rules.

4 Configuration Search

Once the candidate heuristics are produced, MULTI-TAC must find a configuration which works well on the input instances. Each configuration corresponds to an list of search control rules plus the algorithm schema. The space of possible configurations is exponential in the number of candidate heuristics. To search through this space, MULTI-TAC uses a beam search (essentially a parallel hill-climbing search). The beam search takes a beam width B, a set of training instances, and an instance time bound T. The goal is to find the configuration that performs best on the training instances. The best configuration is the one which solves the most instances, given a time bound of T for each instance. If two configurations solve the same number of instances, the one with the least total runtime is preferred. This testing is a type of utility evaluation [6].

Each step in the beam search begins with a set of *parent* configurations. Initially, this set consists of the "empty" configuration, which contains no heuristics. For each parent configuration, the system generates all unique *child* configurations by adding an additional heuristic onto the end of the parent's list. Thus each child configuration includes all of its parent's heuristics, plus one additional heuristic. As alluded to earlier, the ordering on the list is important, because MULTI-TAC prioritizes the rules for each choice point according to their order on the list. For example, if a configuration includes two variable-ordering rules, then the first rule has higher priority; the second rule is used only as a tie-breaker when the first rule does not determine a unique "most-preferred" candidate. We refer to this as a *lexicographic* control strategy.

Each child configuration is tested on the training set, and at the end of each iteration the best B child configurations survive into the next round. These configurations become the parents in the next round, and the process continues until no parent configurations can be improved or the process is interrupted by the user.

We have found that this configuration improvement strategy works well. One reason for its success is due to the lexicographic strategy of employing multiple heuristics. As described earlier, the variable and value ordering heuristics are prioritized, with the first heuristic applied to the set of candidates first and the subsequent heuristics used to break ties in order. During the beam search, configurations are created by choosing the first heuristic, and then adding a second, a third and so on. Since each subsequent heuristic is added to the end of the list, it has a diminishing effect. This "smooths" the search space of configurations, which is exactly the sort of space for which hill climbing is well suited.

The lexicographic ordering scheme also implies that if a child configuration performs worse than its parent, the new heuristic is either giving advice that is worse than arbitrary tie-breaking, which is the default, or is simply too expensive to evaluate; in either case adding more heuristics is not likely to improve the situation. The diminishing return of adding additional heuristics also makes it unlikely that configurations with many heuristics will be particularly good. Therefore, good configurations are more likely to be shorter, which means they will be found early in the search.

4.1 Reducing the Time Required for Utility Evaluation

One drawback to the basic approach to configuration search is that occasionally two (or more) heuristics will interact synergistically, in which case they may be overlooked. For instance, two heuristics that performed quite poorly when tested individually could perform well when used together. Such "surprising pairs" are not likely to be discovered during the search for a good configuration. In particular, we have found that variable-ordering and value-ordering heuristics may tend to interact in this way. To detect these surprising pairs, MULTI-TAC embellishes the beam search by evaluating *all* pairs of heuristics consisting of a single variable selection heuristic and a single value selection heuristic, even if neither

heuristic performs well individually. Unfortunately the number of such pairs can be quite large, making an expensive process even more expensive. This added expense, as well as the cost of configuration search in general, motivates the research presented in this section.

The largest contributor to the cost of configuration search is the utility evaluation. As previously mentioned, for each configuration examined a LISP program is compiled and run on a set of instances. If the program is ill suited to solving the instances, it will continue to run until the instance time bound is reached. This is a "black box" approach to determining if a search control rule is working well. It would be useful if one could estimate how much benefit a search control rule was providing after only a short period of time. This is the focus of some of our current research.

Secondary performance characteristics are behaviors of a problem solver that can be monitored at run-time and used to determine the relative utility of different heuristic methods. For example, we can determine the relative utility of rules approximating Most-Constrained-Variable-First heuristic by estimating the expected cost of using each rule to find a solution. The cost of finding a solution is related to the cost of using the search control rule and the size of the space searched. One of the ways in which Most-Constrained-Variable-First reduces search is by causing the search engine to prune bad paths early. An estimate of the resulting size of the pruned search tree can be produced during run-time, as in [5]. The estimation process works by sampling the branching factor and the depths of the paths explored and the search tree. This data can be collected during backtracking search.

The recorded depth and branching factor information, in conjunction with the cost of using a search control rule, can be used to determine the expected cost of solving the instance. This number can be compared among the several possible operationalizations of Most-Constrained-Variable-First. As a result, the utility evaluation can be performed based on short, partial runs on each instance — runs that are just long enough to make a confident prediction of search cost.

Preliminary results on the accuracy and benefit of using secondary performance characteristics can be found in [1]. Specifically, the paper addresses predicting the best level of constraint propagation in a backtracking framework and predicting the best tabu tenure in an iterative repair framework.

5 Results

In this section we describe describe our experience with one of the problems, MMM(see section 2), on which we performed an in-depth study. Four human subjects participated. One of them was a member of the MULTI-TAC project. The other three were NASA computer scientists who volunteered to participate in our study.[1]

The subjects were asked to write the fastest programs they could. Over a several-day period our subjects spent between 5 and 8 hours designing and implementing their respective programs. The subjects were given access to a "black box" instance generator to test their programs. The instance generator randomly constructed solvable instances of MMM by generating the subset E' and then adding edges.

The instance generator was parameterized, so that it could produce different distributions. We used three different distributions, one for each of the three volunteers. Our fourth subject, the project member, wrote programs for all three of the distributions. As a final comparison, `Simple CSP` records the performance of the backtracking algorithm without any heuristics.

[1]These experiments were done before the inductive method was devised and therefore only used the analytic rule generation method; they are described in more detail in [7]. A comparison of the analytic and inductive approaches to generating search control rules can be found in [10].

	Experiment1		Experiment2		Experiment3	
	CPU sec	unsolved	CPU sec	unsolved	CPU sec	unsolved
MULTI-TAC	4.6	0	27.8	1	449	7
Project member (PM)	3.4	0	76.8	2	1976	33
Subject1	166	6	-	-	-	-
Subject2	-	-	8.9	0	-	-
Subject3	-	-	-	-	1035	7
Simple CSP	915	83	991	98	4500	100

Table 1: Experimental results for three distributions

Table 1 shows the results on 100 randomly-generated instances of the MMM problem. Each of the three experiments in the table refers to a separate distribution. The volunteers generated programs for only a single distribution, so some of the entries in the table are blank. The first two distributions were run with a 10-CPU-second time bound. The third distribution was considerably harder, so we used a 45-CPU-second time bound. In each case, the time bound was established prior to the experiments so that the subjects could design their programs accordingly. MULTI-TAC was trained on each distribution separately (with the appropriate time-bound).

For each experiment, the column labeled "CPU sec" shows, for each program, the cumulative running time for all 100 instances. The second column shows the number of unsolved problems for each program.

The results show that there was considerable variation in the performance of the various programs that were produced. Overall, MULTI-TAC performed on par with our human subjects, and sometimes better. In the first two experiments, the code produced by MULTI-TAC was the second best. In the third experiment, the MULTI-TAC produced the best code. In [7], we discuss the competing programs in some detail: here we will simply summarize the results.

As it turned out, the first two distributions had very similar properties (although the instances in the second distribution were larger), and all the competing programs in the first two experiments had a similar design. On each iteration, an edge was selected to be in the subset E', and backtracking occurred whenever any constraints was violated. The instances were relatively unconstrained and therefore easy to solve. Basically, to get good performance it was sufficient for the programs to statically sort the edges according to the number of neighboring edges. (Other optimizations were helpful, but this was the most important). Subject1 did poorly because he did not include this heuristic. Actually, he reported trying it, but did not experiment with the distribution sufficiently to realize its value.

The third distribution was significantly harder than the first two. For this distribution, the project member used a modified version of his first program with a set of optimizations which did not prove very useful. Subject3 also tried a backtracking approach, but found that an iterative repair method [9] (a local search technique) proved superior.

Interestingly, on this distribution MULTI-TAC produced a backtracking program that was quite different from the program it produced for the first two distributions. Initially, we were surprised at its approach; neither the project members nor the other human volunteers had thought of it. In retrospect, the idea is quite simple. The program used a different value ordering heuristic, so instead of successively growing the subset E', the program was grew the subset $E - E'$, the set of of edges that were *not* in E'. To accompany this value ordering heuristic, the system selected several appropriate variable ordering heuristics. For example, one of the selected heuristics was "choose the edge with the fewest neighbors".

This makes sense, given the value ordering rule, since the edges with the fewest neighbors are the most likely to be included in $E - E'$.

6 Conclusions

In this paper we have given an overview of the MULTI-TAC system, focusing on the different ways it specializes and selects search control heuristics that perform well on a given instance distribution. We also gave a brief outline of our recent work whose goal is to reduce the cost of evaluating competing programs by allowing the programs to be compared after short partial runs. Finally, we described some early experiments showing the performance of programs generated by MULTI-TAC which compared favorably to those generated by humans.

Acknowledgments

The contents of this paper draw heavily from following previously published work: [7, 11, 10]

References

[1] J. A. Allen and S. Minton. Selecting the right heuristic algorithm: Runtime performance predictors. Submitted to the Fourteenth International Joint Conference on Artificial Intelligence.

[2] D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.

[3] G. F. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2), 1986.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., 1979.

[5] D. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29:121–136, 1975.

[6] S. Minton. *Learning Search Control Knowledge: An Explanation-based Approach.* Kluwer Academic Publishers, Boston, Massachusetts, 1988. Also available as Carnegie-Mellon CS Tech. Report CMU-CS-88-133.

[7] S. Minton. Integrating heuristics for constraint satisfaction problems: A case study. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, San Jose, CA, 1993. AAAI Press.

[8] S. Minton, M. Johnston, A.B. Philips, and P. Laird. Solving large scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings AAAI-90*, 1990.

[9] S. Minton, M. Johnston, A.B. Philips, and P. Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[10] S. Minton and I. Underwood. Small is beautiful: A brute-force approach to learning first-order formulas. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, WA, 1994. AAAI Press.

[11] S. Minton and S. Wolfe. Using machine learning to synthesize search programs. In *Proceedings of the Ninth Conference on Knowledge-Based Software Engineering*, 1994.

[12] T. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-base generalization: A unifying view. *Machine Learning*, 1(1), 1986.

[13] D.R. Smith. KIDS: A knowledge-based software development system. In M.R. Lowry and R.D. McCartney, editors, *Automating Software Design*. AAAI Press, 1991.

Incorporating Efficient Operations Research Algorithms in Constraint-Based Scheduling

Philippe Baptiste and Claude Le Pape and Wim Nuijten

ILOG S.A., 2 Avenue Gallieni, BP 85, F-94253 Gentilly Cedex, France
Email: {lepape, nuijten}@ilog.fr
Url: http://www.ilog.fr

Abstract

We address the area of scheduling and the differences between the way operations research and artificial intelligence approach scheduling. We introduce the concept of constraint programming, and describe how operations research techniques can be integrated in constraint programming. Finally, we give a short overview of the results obtained with our approach.

1 Introduction

Baker [1974] defines scheduling as the problem of allocating scarce resources to activities over time. Scheduling problems arise in areas as diverse as production planning, personnel planning, computer design, and time tabling. Over the years, the theory and application of scheduling has grown into an important field of research, and an extensive body of literature exists on the subject. For more elaborate introductions to the theory of scheduling, we refer to [Baker, 1974], [Coffman, 1976] and [French, 1982].

Roughly speaking, we can distinguish two fields of research that pay attention to scheduling, viz., *operations research* (OR) and *artificial intelligence* (AI). Traditionally, a lot of the attention in OR has been paid to scheduling problems that are based on relatively simple mathematical models. For solving the problem at hand, the combinatorial structure of the problem is heavily exploited, leading to improved performance characteristics. We could say that an OR approach often aims at achieving a high level of *efficiency*

in its algorithms. However, when modeling a practical scheduling problem using these classical models, one is often forced to discard many degrees of freedom and side constraints that exist in the practical scheduling situation. Discarding degrees of freedom may result in the elimination of interesting solutions, regardless of the solution method used. Discarding side constraints gives a simplified problem and solving this simplified problem may result in impractical solutions for the original problem.

In contrast, AI research tends to investigate more general scheduling models and tries to solve the problems by using general problem solving paradigms. We could say an AI approach tends to focus more on the *generality of application* of its algorithms. This, however, implies that AI algorithms may perform poorly on specific cases, compared to OR algorithms.

So, on the one hand we have OR which offers us *efficient* algorithms to solve problems that however might not be well suited to be used in practice, and on the other hand we have AI that offers us algorithms that are more *generally applicable*, but that might suffer from somewhat poor performance. Naturally, we want the best of both worlds, i.e., we want efficient algorithms that we can apply to a wide range of problems. In this article, we introduce *constraint programming* which provides a general modeling and problem solving paradigm, and we show how to integrate it with OR algorithms, in order to improve performance of the approach.

2 Constraint Programming

Generally speaking, constraint programming is concerned with solving instances of the *Constraint Satisfaction Problem* (CSP). An instance of the CSP consists of a set of *variables*, a *domain* for each variable specifying the values to which the variable may be assigned, and a set of *constraints* on the variables. A constraint denotes a relation between the values of one or several variables. For instance, if x is an integer variable, $x < 10$ is a constraint on the variable x. Solving an instance of the CSP consists in assigning values to variables such that all constraints are satisfied simultaneously.

In constraint programming, constraints are exploited to reduce the amount of computation needed to solve the problem. More specifically, they are used to reduce the domains of the variables and to detect inconsistencies. This deductive process is called *constraint propagation*.

For example, from $x < y$ and $x > 8$, we deduce, if x and y denote

integers, that the value of y is at least 10. If later we add the constraint $y \leq 9$, a contradiction is immediately detected. Without propagation, no contradiction would be detected until the instantiation of both x and y.

For complexity reasons, constraint propagation is usually *incomplete*. This means that some but not all the consequences of constraints are deduced. In particular, constraint propagation cannot detect all inconsistencies. Consequently, heuristic search algorithms must be implemented to explore possible refinements of the instance of the CSP. e.g., by assigning a value to each variable. and exhibit solutions that are guaranteed to satisfy the constraints.

In the remainder of this article, Sections 3 and 4 discuss the propagation of two types of constraints that occur in scheduling problems, viz., *temporal constraints* and *resource constraints.* Furthermore, Section 5 presents an overview of the results we obtained with our approach. Finally, Section 6 presents some conclusions.

3 Temporal Constraints

A temporal constraint is logically expressed by a formula $I_i + d \leq I_j$. I_i and I_j are time points and d a delay that must elapse between these time points. Generally, a time point corresponds to the beginning or the end of an activity. If A and B are two activities, and if a time unit corresponds to a minute, the temporal constraint $end(A) + 5 \leq start(B)$ means that at least 5 minutes must pass between the end of A and the beginning of B. A special time point O is often defined to represent the origin of time: $O + 60 \leq start(A)$ means that A cannot start before date 60; $end(A) - 120 \leq O$ means that A cannot end after date 120.

Ford's algorithm [Ford, 1956; Gondran & Minoux, 1984], which computes the length of the longest paths from a node N_0 to the other nodes N_1 ... N_n of a valued oriented graph, can be used to compute the earliest start and end times of activities. Informally, this algorithm can be described as follows.

1. Let $\pi(N_0) = 0$ and $\pi(N_i) = -\infty$ for every i between 1 and n.
2. For every j between 1 and n, replace $\pi(N_j)$ with the maximum of $\pi(N_j)$ and $\max(\pi(N_i) + l_{ij})$ where l_{ij} denotes the length of the arc going from N_i to N_j when this arc exists.
3. Iterate step 2 until $\pi(N_i)$ becomes stable for every i.

This algorithm can also be used to compute the latest start and end times of

activities by exploring the graph in the reverse direction. When the temporal constraints are globally compatible, it updates the time-bounds of activities (earliest and latest start and end times) in $O(mn)$ where m is the number of constraints and n the number of activities. Note that one can easily implement an incremental version of the algorithm. Indeed, at each iteration, it is useless to re-compute $(\pi(N_i)+l_{ij})$ if the preceding iteration did not result in an update of $\pi(N_i)$. Consequently, the algorithm can be modified to make the update of $\pi(N_i)$ trigger, at the next iteration, the update of $\pi(N_j)$.

In a constraint programming framework, the constraint $I_i + d \leq I_j$ can be propagated in a very simple way. Each time the minimal value of I_i changes (becomes I_i^{min}), the minimal value of I_j is set to the maximum of $I_i^{min} + d$ and the current minimal value of I_j. In the reverse direction, each time the maximal value of I_j changes (becomes I_j^{max}), the maximal value of I_i is set to the minimum of $I_j^{max} - d$ and the current maximal value of I_i. The main advantage of the constraint programming implementation is that each constraint is considered separately. If other constraints exist, the algorithm that performs the propagation propagates also the other constraints. Other constraints can consequently be designed independently of the temporal constraints.

An issue that arises is the efficiency of a constraint programming implementation, compared to Ford's algorithm. It is shown in [Le Pape, 1988] that the constraint propagation steps can be ordered to obtain the same complexity as Ford's algorithm when the temporal constraints are globally compatible. In particular, the constraint propagation method which consists in using a first-in first-out queue of constraints updates the time-bounds of activities in $O(mn)$. This method is used in ILOG SCHEDULE [Le Pape, 1994], an add-on to ILOG SOLVER, a C++ library for constraint programming [Puget, 1994].

4 Resource Constraints

In scheduling problems, activities require resources which are available in finite amounts over time. Three main classes of resources can be distinguished:

- *Unary resources*: a unary resource is a resource of capacity one (e.g. a specific person). Two activities that require the same unary resource cannot overlap.
- *Volumetric resources*: a volumetric resource typically represents a pool of many non-differentiated resources (e.g. a group of people with the

same capabilities). At any point in time, the number of units required by the executing activities cannot exceed the number of units that are available.

- *State resources*: a state resource is a resource that can be used for an activity only when it is in a given state. Two activities that require the same resource in different states cannot overlap.

There does not appear to exist algorithms as general as Ford's algorithm to optimally update the time-bounds of activities submitted to resource constraints. As a matter of fact, the problem of eliminating all the impossible start and end times of activities submitted to resource constraints is *NP-hard* [Garey & Johnson, 1979]. Even when only one unary resource is considered, the problem of determining whether there exists a schedule satisfying given time-bounds for each activity is NP-hard [Garey & Johnson. 1979]. As a result, many OR algorithms have been developed to compute time-bounds for specific problems. Each of these algorithms corresponds to a different tradeoff between the generality of the algorithm, i.e., the class of problems to which the algorithm applies, the precision of the computed time-bounds, and the computational complexity of the algorithm.

One of the most successful OR algorithms for updating time-bounds of activities submitted to unary resource constraints was proposed by Carlier & Pinson [1990]. The main principle of this algorithm is to compare the temporal characteristics of an activity A to those of a set of activities Ω which require the same resource. Let est_A denote the earliest possible start time of A, let_A the latest possible end time of A, and p_A the processing time of A. Let est_Ω denote the smallest of the earliest start times of the activities in Ω, let_Ω denote the greatest of the latest end times of the activities in Ω, and p_Ω denote the sum of the processing times of the activities in Ω. The following rules apply:

$$\left[\begin{array}{l} let_\Omega - est_\Omega < p_A + p_\Omega \\ let_A - est_\Omega < p_A + p_\Omega \end{array}\right] \Rightarrow [A \textit{ is before all activities in } \Omega]$$

$$\left[\begin{array}{l} let_\Omega - est_\Omega < p_A + p_\Omega \\ let_\Omega - est_A < p_A + p_\Omega \end{array}\right] \Rightarrow [A \textit{ is after all activities in } \Omega]$$

New time-bounds can consequently be deduced. When A is before all activities in Ω, the end time of A is necessarily at most $let_\Omega - p_\Omega$. When A is after all activities in Ω, the start time of A is necessarily at least $est_\Omega + p_\Omega$.

The technique which consists in applying these rules is known as *edge-finding*. Notice that if n activities require the resource, there are potentially $O(n * 2^n)$ pairs (A, Ω) to consider. An algorithm that performs all of the possible time-bound adjustments in $O(n^2)$ is presented in [Carlier & Pinson, 1990]. Impressive results have been obtained by applying variants of this algorithm as part of a tree search procedure to the Job Shop Scheduling Problem (JSSP) as defined in [Garey & Johnson, 1979]. Examples of such approaches are [Carlier & Pinson, 1990], [Applegate & Cook, 1991], and [Carlier & Pinson, 1994]. However, as in the case of temporal constraints, it is unclear what ought to be done when a problem requires the satisfaction of other types of constraints, in addition to temporal and unary resource constraints. Clearly, Carlier & Pinson's algorithm could still be useful, but a specific implementation may not be easy to integrate with the other components needed to solve the problem.

This contrasts with the work done in the field of AI. In this field, various classes of resource constraints are naturally integrated in a global framework that incorporates other types of constraints as well. Two main mechanisms are used to propagate resource constraints.

- The first mechanism uses *time-tables* to maintain information about the variations of resource utilization and resource availability over time. Resource constraints are propagated in two directions: from the resources to the activities, in order to update activity time-bounds according to the availability of the resources; from the activities to the resources, in order to update resource utilization and availability according to the time-bounds of activities. For example, when the latest start time *lst* of an activity is smaller than its earliest end time *eet*, it is sure that the activity will use the resource between *lst* and *eet*. Over this period, the corresponding resource amount is no longer available for other activities. In ILOG SCHEDULE [Le Pape, 1994], time-tables can be used to represent unary, volumetric, and state resources. They also enable to state that at least some amount of resource capacity must be used over a given period.
- The second mechanism consists of posting disjunctive constraints. The most basic disjunctive constraint states that two activities A and B that require the same unary resource R cannot overlap in time: either

A precedes B or B precedes A. This can be written as

$$end(A) \leq start(B) \ \vee \ end(B) \leq start(A).$$

Constraint propagation consists in reducing the set of possible values for the start and end variables: whenever the smallest possible value of $end(A)$ exceeds the greatest possible value of $start(B)$, A cannot precede B; hence B must precede A; the time-bounds of A and B can consequently be updated with respect to the new temporal constraint $end(B) \leq start(A)$. Similarly, when the earliest possible end time of B exceeds the latest possible start time of A, B cannot precede A. When neither of the two activities can precede the other, a contradiction is detected. This way of propagating constraints enforces arc-B-consistency as defined by Lhomme [1993]. Several extensions of the basic disjunctive constraint are discussed in [Baptiste & Pape, 1995]. This includes (1) activities that may or may not require the resource, (2) disjunctive constraints applied to state resources, and (3) setup times between activities that require the same resource. These extensions are all provided in ILOG SCHEDULE.

A few issues are raised by the integration of algorithms such as the edge-finder of Carlier & Pinson in a generic constraint propagation framework.

- First, it is necessary to identify the events that should trigger the execution of the algorithm. When activities are certain to use the resource and their durations are fixed, only the modification of the earliest start and latest end times of the activity can lead the edge-finder to deduce more precise time-bounds. When the processing time of an activity is itself a constrained variable, the situation is different: to deduce only correct information, the edge-finder must rely on the smallest possible duration of the activity; when the smallest possible duration increases, the edge-finder must be called again to ensure that all the possible adjustments are done.
- Second, efficiency issues have to be considered. Indeed, when both temporal constraints and resource constraints apply, it is intuitively more efficient to deduce all the consequences of temporal constraints prior to apply the edge-finder. The main reason for that is that the propagation of each temporal constraint has a very low cost in comparison to an application of the edge-finder. The edge-finder is consequently delayed until the other constraints have been propagated.

- Implementation issues also have to be considered. In the context of an incremental constraint satisfaction framework, it must be possible to add a new activity at any stage of the problem-solving process. This modifies the data structures that can be used as a basis for the implementation of the edge-finder.

A variant of the edge-finder of Carlier & Pinson [1990] is presented in [Nuijten, 1994]. This algorithm has the same complexity but has a much simpler structure. It, furthermore, can easily be generalized to be used for instance for volumetric resources, as is shown in [Nuijten, 1994]. There, the algorithm is already integrated in a constraint-based approach.

Both the [Carlier & Pinson, 1990] and the [Nuijten, 1994] versions were tested on unary resources in the context of ILOG SCHEDULE. We remark that Carlier & Pinson [1994] presents a variant running in $O(n * log(n))$ that we have not tested yet. The experimental results presented in [Baptiste, 1994] show that the two $O(n^2)$ versions are roughly equivalent in terms of CPU time. They also show edge-finding to be extremely powerful in comparison to arc-B-consistency. As the [Nuijten, 1994] version was simpler, it was retained and definitely implemented in version 1.1 of ILOG SCHEDULE. This allows the users of ILOG SCHEDULE to enjoy the efficiency of the edge-finder in the flexible context of constraint programming. In particular, the use of the edge-finder does not prevent the user from defining activities of initially unknown duration, activities that may or may not require the resource, setup times between activities, or any other user-defined constraint.

5 Computational Results

This section presents some of the results we obtained by incorporating OR algorithms in constraint-based approaches to scheduling.

5.1 Job Shop Scheduling Approximation

Nuijten [1994] presents an approximation approach based on constraint satisfaction techniques which is applied to a number of scheduling problems amongst which the JSSP and a generalization of the JSSP in which machines can have an arbitrary capacity, called the Multiple Capacitated Job Shop Scheduling Problem (MCJSSP) [Nuijten & Aarts, 1994].

The results on these two problems are good. Out of 43 well known instances of the JSSP, 31 instances are solved to optimality, including the notorious MT10 instance, and the deviation of the minimum found makespan

from the best known lower bound is on average 0.72%. Vaessens, Aarts & Lenstra [1994] compare 20 of the best approaches to the JSSP for 13 instances of the aforementioned set of 43 instances. Ranking the approaches that were used according to effectiveness, this approach comes in the sixth place with an average deviation of 2.26%, where the *tabu search* approach of Nowicki & Smutnicki [1993] performs best with an average deviation of 0.54%. The results on the MCJSSP are also good; out of 30 instances, 22 are solved to optimality and the deviation of the upper bounds from the lower bounds on the minimal makespan is on average 0.52%. For more extensive results we refer to [Nuijten, 1994].

5.2 Job Shop Scheduling Optimization

We also devised an optimization algorithm for the JSSP [Baptiste, 1994]. This algorithm is based on Branch-and-Bound backtracking search with constraint propagation being performed at each node of the search tree. It was tested on more than eighty instances. Table 1 gives the results obtained on the ten 10x10 JSSP instances used by Applegate & Cook in their computational study of the JSSP [Applegate & Cook, 1991]. In Table 1, columns "BT" and "CPU" give the total number of backtracks and CPU time needed to find an optimal solution and prove its optimality. Columns "BT(pr)" and "CPU(pr)" give the number of backtracks and CPU time needed for the proof of optimality. Column "CPU(1)" gives the CPU time needed to find the first solution, including the time needed for stating the problem and performing initial constraint propagation. Column "TM" gives the total amount of memory used to represent and solve the problem (in kilobytes). All CPU times are given in seconds on a HP715/50 workstation.

ILOG SCHEDULE performs better or about as well as the specific procedure of Applegate & Cook [1991] on five problems in terms of CPU time, and on seven problems in terms of the number of backtracks needed to solve the problem. Over the ten problems, the total number of backtracks for ILOG SCHEDULE is 579711, while the total number of nodes explored by Applegate & Cook's algorithm is 674128. The integration of an edge-finder within ILOG SCHEDULE allows its users to enjoy the flexibility inherent to constraint programming, with performance in the same range of efficiency as specific OR algorithms.

Instance	CPU(1)	BT	CPU	BT(pr)	CPU(pr)	TM
MT10	.6	69758	1916.3	7792	230.6	140
ABZ5	.6	17636	401.7	5145	115.0	136
ABZ6	.5	898	27.3	291	8.4	136
LA19	.6	21910	529.6	5618	137.9	140
LA20	.6	74452	1521.0	22567	443.8	136
ORB1	.7	13944	412.6	5382	165.4	144
ORB2	.6	114715	3552.3	30519	927.4	140
ORB3	.6	190117	5597.0	25809	770.7	140
ORB4	.5	64652	2004.1	22443	701.8	140
ORB5	.6	11629	303.6	3755	96.8	140

Table 1: Results on ten 10x10 instances of the JSSP

5.3 A Practical Problem

The edge-finder was also tested on an industrial project scheduling problem submitted by a customer of ILOG, and found to be extremely difficult to solve. The problem consists of scheduling two projects that require common resources. There are 45 activities and five resources to consider. Each activity requires up to four resources. Within each project, activities are subjected to precedence constraints. Three of the resources are unary resources. The number of activities that require each unary resource is close to 30. The two other resources are volumetric resources with capacity greater than one.

There are two optimization criteria, viz., minimization of the end times of two specific activities, one for each project. As the projects rely on common resources, these two optimization criteria are conflicting. As a result, the final user wants to impose upper bounds on the two criteria, and wants the system to tell whether there exists a solution satisfying these upper bounds.

Table 2 reports the CPU times, in seconds, obtained for different values of the upper bounds of the two criteria. We compared two algorithms, one using arc-B-consistency and one using the edge-finder of Nuijten [1994]. If an algorithm is incapable of solving an instance within one hour of CPU time, that is reported by "–". Solving an instance means either finding a solution or proving that there is no solution. Numbers in bold correspond to the upper bounds for which there is no solution. Table 2 clearly shows that the edge-finder strongly outperforms arc-B-consistency.

	Algorithm	120	125	130	134	135	136	140
120	arc-B-consistency	**6**	**39**	**131**	**805**	**1188**	621	1
	edge-finder	**1**	**1**	**1**	**1**	**31**	1	1
125	arc-B-consistency	**40**	**387**	**1771**	–	–	–	1
	edge-finder	**1**	**1**	**1**	**1**	**50**	1	2
130	arc-B-consistency	**100**	**1172**	–	–	–	–	1
	edge-finder	**1**	**1**	**1**	**1**	**117**	2	2
134	arc-B-consistency	**276**	–	–	–	–	–	1
	edge-finder	**1**	**1**	**1**	**1**	**215**	2	2
135	arc-B-consistency	**356**	–	–	–	–	–	1
	edge-finder	**10**	**58**	**171**	**265**	**299**	113	2
136	arc-B-consistency	247	–	–	–	–	–	1
	edge-finder	1	1	2	2	2	2	2
140	arc-B-consistency	1	1	1	1	1	1	1
	edge-finder	1	1	1	2	2	2	2

Table 2: Results on an industrial project scheduling problem

6 Conclusions

We have shown how to combine operations research and artificial intelligence, in particular constraint programming, in a way that preserves the best of both, i.e., we preserved the efficiency provided by operations research and the generality of approach offered by constraint programming. We have shown that a good performance can be obtained on such a classical scheduling problem as the JSSP as well as on generalizations thereof, and on real-life scheduling problems as described in Section 5.3. In short, we think that in ILOG SCHEDULE we have found a powerful combination of techniques that allows us to tackle a broad range of practical scheduling problems in an efficient way.

References

APPLEGATE, D., AND W. COOK [1991], A computational study of the job-shop scheduling problem, *ORSA Journal on Computing* **3**, 149–156.

BAKER, K.R. [1974], *Introduction to Sequencing and Scheduling*, Wiley & Sons.

BAPTISTE, P. [1994], Constraint-based scheduling: Two extensions, Master's thesis, University of Strathclyde.

BAPTISTE, P., AND C. LE PAPE [1995], Disjunctive constraints for manufacturing scheduling: Principles and extensions, to appear.

CARLIER, J., AND E. PINSON [1990], A practical use of Jackson's preemptive schedule for solving the job shop problem, *Annals of Operations Research* **26**, 269–287.

CARLIER, J., AND E. PINSON [1994], Adjustment of heads and tails for the job-shop problem, *European Journal of Operational Research* **78**, 146–161.

COFFMAN, JR., E.G. (ed.) [1976], *Computer & Job Shop Scheduling Theory*, Wiley, New York.

FORD, JR., L.R. [1956], *Network Flow Theory*, Technical report, Rand Corporation.

FRENCH, S. [1982], *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, Wiley & Sons.

GAREY, M.R., AND D.S. JOHNSON [1979], *Computers and Intractability: A Guide to the Theory of NP-completeness*, W.H. Freeman and Company, New York.

GONDRAN, M., AND M. MINOUX [1984], *Graphs and Algorithms*, John Wiley and Sons.

LE PAPE, C. [1988], *Des systèmes d'ordonnancement flexibles et opportunistes*, Ph.D. thesis, University Paris XI, in French.

LE PAPE, C. [1994], Implementation of resource constraints in ILOG SCHEDULE: A library for the development of constraint-based scheduling systems, *Intelligent Systems Engineering* **3**, 55–66.

LHOMME, O. [1993], Consistency techniques for numeric CSPs, *Proc. 13th International Joint Conference on Artificial Intelligence*.

NOWICKI, E., AND C. SMUTNICKI [1993], *A Fast Taboo Search Algorithm for the Job Shop Problem*, Preprinty nr. 8/93, Instytut Cybernetyki Technicznej, Politechnicki Wroclawskiej, Poland.

NUIJTEN, W.P.M., AND E.H.L. AARTS [1994], Constraint satisfaction for multiple capacitated job shop scheduling, in: A. Cohn (ed.), *Proc. 11th European Conference on Artificial Intelligence*, John Wiley & Sons, 635–639.

NUIJTEN, W.P.M. [1994], *Time and Resource Constrained Scheduling: A Constraint Satisfaction Approach*, Ph.D. thesis, Eindhoven University of Technology.

PUGET, J.-F. [1994], *A C++ Implementation of CLP*, Technical Report 94-01, ILOG, S.A., Gentilly, France.

VAESSENS, R.J.M., E.H.L. AARTS, AND J.K. LENSTRA [1994], *Job Shop Scheduling by Local Search*, COSOR Memorandum 94-05, Eindhoven University of Technology.

Position Paper for First International Joint Workshop on Artificial Intelligence and Operations Research

Thomas Dean
Department of Computer Science
Brown University
115 Waterman Street
Providence, RI 02906, USA
Phone: (401) 863-7600
Fax: (401) 863-7657
Email: tld@cs.brown.edu

1 Introduction

My work over the last ten years has been concerned with automated planning, learning system dynamics, adaptive control, and Markov decision processes. The focus is on dynamical systems that can be represented as discrete-time, discrete-space, stochastic processes. My current emphasis is on solving problems posed as Markov decision process with very large state spaces.

My students and I have developed methods whereby the underlying dynamical systems can be efficiently represented and have designed and analyzed algorithms for efficiently solving restricted classes of problems. We have worked with computer scientists, operations researchers, and applied mathematicians in seeking better solution methods, *e.g.,* Craig Boutilier and Marty Puterman at the University of British Columbia, Moises Goldszmidt at Rockwell Science Center, Leslie Kaelbling and Harold Kushner at Brown University, Stuart Russell at the University of California at Berkeley, Michael Wellman at the University of Michigan, and Steve Hanks at the University of Washington. Together we are building a body of shared knowledge from diverse fields aimed at solving an important class of decision problems.

Mike Wellman and I attempted a synthesis of ideas from operations research (OR) and artificial intelligence (AI) that pertain to planning and control under uncertainty (*Planning and Control*, Morgan-Kaufmann, 1991). Stuart Russell, Keiji Kanazawa, Daphne Koller (all from the University of California at Berkeley) are working on a text that focuses on the connection between Markov decision processes and decision-theoretic methods for automated planning. In the following, I provide a high-level perspective on our work, mention specific research projects, suggest some concrete proposals for participation in the workshop, and provide pointers to abstracts and postscript of recent papers available on the world wide web.

2 Perspective

Logic provides a very general representation language and, at least for the propositional case, solving boolean satisfiability (SAT) problems provides the corresponding general class of computational problems. One property of such problems is that the inputs (boolean formula) and outputs (satisfying assignments) can be compactly represented. The interesting computational issues revolves around searching the exponential number of possible assignments to find a satisfying one.

Markov decision processes (MDPs) provide general semantic foundations for a large class of automated planning problems. MDPs can be represented as either SAT problems, in the case of deterministic processes based on state-space operators, or as linear programs (LPs), in the case of stochastic variants with separable value functions. However, time (dynamics) provides additional structure that can. in many cases, serve to expedite computations. Dynamic programming is one example of a general method that allows us to exploit the structure in dynamics and separable value.

As in the case SAT problems, we are interested in classes of MDPs in which the problem descriptions (cost and state-transition functions) and solutions (functions mapping states to actions) can be encoded in space some low-order polynomial of the number of state variables.[1] Not too surprisingly, even this restricted class of problems includes some vary hard problems.

Traditionally, researchers in OR have been interested in proving convergence to an optimal solution for various sorts of iterative algorithms. Convergence in the limit, while a very nice property, is neither necessary nor sufficient for good performance. Researchers in AI and computer science are interested in proving deterministic performance criteria of the form, this algorithm will return a solution whose value is within $(1-\epsilon)$ of the optimal solution in time polynomial in the size of the problem specification and $1/\epsilon$, and probabilistic performance criteria of the form, with probability $1-\delta$, this algorithm will return a solution whose value is within $(1-\epsilon)$ of the optimal solution in time polynomial in the size of the problem specification, $1/\epsilon$, and $1/\delta$. An interesting question is for what subclasses of problems do algorithms satisfying the above criteria exist and how might we specify such subclasses in a mathematically concise manner.

The machinery to state and answer such questions is now becoming available. We have developed approximation algorithms and proved such performance guarantees for a range of system identification problems and are now turning our attention to Markov decision processes. There is now a chance to revisit some of the classic problems of OR and provide better algorithms and more appropriate forms of analysis.

An effective study of these questions will require the collaboration of researchers in both AI and OR and initial collaborations have begun. On the OR side, researchers like Dimitri Bertsekas. George Dantzig, Harold Kushner, Martin Puterman, and Sheldon Ross are beginning to take an interest in AI problems and techniques. Issues at the frontier of automated planning and computational learning theory are being explored by researchers in OR and computer science including Christos Papadimitriou, Michael Luby, and Manfred Warmuth. We have contributed to this research both in terms of concrete results both theo-

[1] In the case of some AI planning problems, state variables are often called fluents and correspond to propositions whose truth values change over time.

retical and empirical and in terms of consciousness raising and promoting cross-disciplinary collaborations.

3 Recent Research Focus

Here are three basic areas of research in which my students and I are active and we believe are ripe for concerted cross-disciplinary effort.

1. Designing approximation algorithms with provable performance guarantees. In particular, we believe that recent work in competitive algorithm analysis and computational learning theory will allow us to design stopping criteria for algorithms such as value iteration and modified policy iteration that are guaranteed to generate a solution that is within ϵ of the optimum with probability $1 - \delta$ in time that is some low-order polynomial of the size of the problem, $1/\epsilon$, and $1/\delta$.

2. We are developing new aggregation algorithms for computing optimal policies that operate on state spaces that cannot be explicitly enumerated (the state space is of size $O(2^M)$ for some reasonably large M) and yet have problem representations and solutions that can be stored in $O(M)$.

3. We are working on decomposition algorithms for solving MDPs with very large state spaces using iterative approximation methods that can be shown to converge to optimal solutions. These iterative methods are motivated by heuristic divide-and-conquer techniques in AI, but the iterative schemes can be justified in terms of shadow prices (Lagrange multipliers) and Dantzig-Wolfe decomposition methods for solving large linear programs.

4 Participation

I would be happy to give a tutorial on Markov decision processes and decision theoretic planning. I have given similar tutorials to a wide range of audiences and would find it relatively easy to tailor my lecture to a mixed AI and OR audience. Relevant postscript files[2] include

- `DeanSIPRAI-94.tutorial.ps`,
- `DeanSIPRAI-94.slides.one.ps`, and
- `DeanSIPRAI-94.slides.two.ps`.

I would also be willing to give a presentation concerning the use of decomposition schemes for solving very large Markov decision processes with quick introductions to factored state-space representations, solving MDPs using linear and dynamic programming, and Dantzig-Wolfe decomposition theory. Relevant postscript files include,

[2]All of the postscript files mentioned in this position paper are found in `ftp://cs.brown.edu/u/tld/postscript/`.

- `DeanKaelblingKirmanandNicholsonAIJ-95.ps`, and
- `DeanandLinIJCAI-95.submitted.ps`.

5 Recent Papers

On the web,[3] you will find a dozen or so papers and abstracts including the following. Three recent papers (and their abstracts) submitted to IJCAI on MDPs and automated planning. One AAAI-1993 paper and one paper to appear in Artificial Intelligence on aggregation methods for solving MDPs. A tutorial and slides on Markov decision processes presented at the 1994 Conference on Uncertainty in AI and at the 1993 Summer Institute on Probability and AI. Information on a 1991 book entitled *Planning and Control* (with Mike Wellman) that uses MDPs and notions from control theory to present a unified view of planning under uncertainty.

[3] URL: `http://www.cs.brown.edu/people/tld/home.html`

Robust Encodings of OR Problems for Genetic Algorithms

James C. Bean, Atidel Hadj-Alouane and Bryan Norman

University of Michigan
Department of Industrial and Operations Engineering
Ann Arbor, MI 48109-2117

January 24, 1995

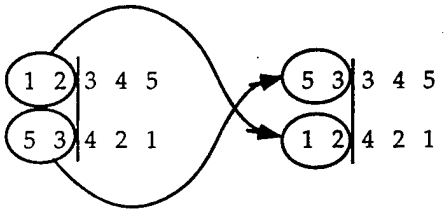
Genetic algorithms were developed in computer science in the mid 60's by Holland [1975]. They seek to breed good solutions to complex problems by a paradigm that mimics evolution. A population of solutions is constructed. Solutions in the population mate and bear offspring solutions in the next generation. These "reproduction" and "crossover" operations are programmed to replicate the paradigm of "survival-of-the-fittest." Over many generations the solutions in the population improve until the best of the population is (hopefully) near optimal. For background on traditional genetic algorithms the reader is referred to Holland [1975] and Goldberg [1989].

Adopting the basic terminology of genetics: a *chromosome* is an encoding of a solution and is a vector in $\Re^n$; a *gene* is an element of the chromosome (vector); an *allele* is a value taken by that element. For example, $x \in \Re^9$ might be a chromosome, x_4 one of its genes, and if $x_4 = 3.5$ then the fourth gene has allele 3.5.

Crossover is the process by which elements of two parent chromosomes recombine to create a new offspring chromosome. The traditional crossover operator is the one-point crossover. To illustrate, consider a simple genetic algorithm approach to the single machine sequencing problem. A candidate solution to a single machine sequencing problem is an ordering of the n jobs. Two such orderings for five jobs are $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$ and $5 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 1$. The most direct chromosomal representation of such sequences are the permutations $x = (1, 2, 3, 4, 5)$ and $x' = (5, 3, 4, 2, 1)$. A one-point crossover operation would cleave each permutation at some point, say after the second job in the sequence, and exchange leading segments. Executing that process on the example schedules gives $5 \rightarrow 3 \rightarrow 3 \rightarrow 4 \rightarrow 5$ and $1 \rightarrow 2 \rightarrow 4 \rightarrow 2 \rightarrow 1$ (see Figure 1).

Neither is a valid sequence. Genetic algorithms have been slow to gain acceptance for operations research problems since crossing over two feasible solutions does not, in many cases, result in a feasible solution as an offspring.

FIGURE 1: One-point Crossover



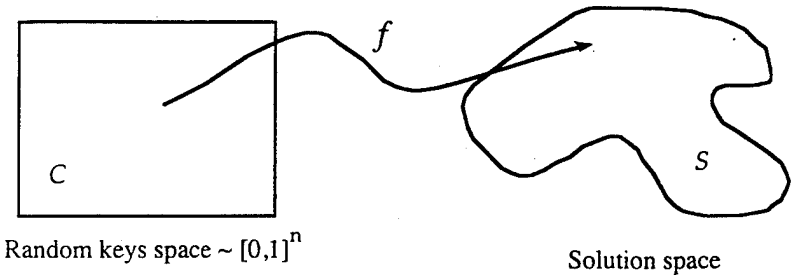
Many authors have developed problem specific representations of solutions for scheduling problems that overcome the offspring feasibility difficulty. Genetic algorithms have been applied to scheduling problems by Syswerda [1991]; Kanet and Sridharan [1991]; Biegal and Davern [1990], Storer et al. [1992a, 1992b], Herrmann and Lee [1993], Lee and Piramuthu [1994] and others. Two continuing drawbacks have been the need for specialized representations for each problem variation and the experimentation necessary to set parameters in each representation. Our work concentrates on robust methods for encoding operations research problems so that general use crossover operators lead to feasible solutions. Two approaches have been developed: random keys and multiple-choice. We mention both here, but concentrate on the latter.

Random Keys (Bean [1994], Norman and Bean [1994]) uses a paradigm similar to that of a random variable. A random variable is a function from the outcome space of a real stochastic phenomenon to the real line. Many stochastic phenomena map, by various random variables, to the same distribution in $\Re$. For example, tossing a coin can be represented by the random variable x where $x(H) = 1, x(T) = 0$, with a given probability, p, of heads. A machine that can be up or down can be modeled by the random variable y where $y(U) = 1, y(D) = 0$ with a given probability, p, of the machine being up. Both phenomena reduce to the Bernoulli distribution. Now, we can analyze the Bernoulli distribution for expectation, variance, etc. and, in effect, analyze many real stochastic phenomena.

We wish to establish a space, C, analogous to $\Re$ for a random variable, such that the feasible regions of many operations research problems correspond to C. If a single genetic algorithm is developed to search C efficiently, we can solve a multitude of operations research problems with this algorithm. For each problem we need only redefine the mapping between the original space

and the simpler random keys space. See Figure 2 for a graphical depiction of the process.

FIGURE 2: Random Keys and Solution Spaces



The idea of a surrogate space is not new, having been used for years by Syswerda XXX, Storer et al. [1992a,1992b], Herrmann and Lee [1993] and others. The unique facet of random keys is that the same space is used for many problems allowing a stable genetic algorithm to be used for each of those problems.

The random keys approach is most effective on sequencing type problems such as scheduling and vehicle routing. It has also been used on resource allocation and quadratic assignment problems. An alternative approach, the multiple-choice encoding, is designed for general integer programs.

The difficulty with constructing a genetic algorithm for integer programming is the general constraint set. Given two feasible solutions as parents, it is difficult to guarantee that the offspring resulting from a crossover are also feasible. A common approach to this problem is the use of penalty functions (Michalewicz and Janikow [1994]) to relax troublesome constraints and penalize the objective for violating them. Approaches in the literature have had limited success and may result in substantial solution error due to the relaxation.

The multiple-choice encoding is based on a nonlinear penalty function that results in a strong dual (Hadj-Alouane and Bean [1992] and Bean and Hadj-Alouane [1992]). That is, parameters for the penalty function exist such that the relaxed problem solves the original problem exactly. Given a fixed parameter vector, we solve a nonlinear integer program with a genetic algorithm. To find the appropriate parameter vector, a search of the parameter space is carried out with a simplistic subgradient-type algorithm. The combination is a primal-dual genetic algorithm for general integer programming.

Computational tests have been quite successful for a range of problems including multiple-choice integer programs, general linear integer programs, nonconvex integer programs and assembly line balancing problems.

As an example, consider the multiple-choice integer program:

$$\begin{aligned} &\min\ cx \\ \text{subject to}: \quad &Ax \geq b \\ &\sum_{j=1}^{n_i} x_{ij} = 1, \quad i = 1, 2, \ldots, m \\ &x \geq 0, \text{ integer.} \end{aligned} \tag{$MCIP$}$$

Define the penalty function $p_\lambda(x) = \sum_k \lambda_k [\min(0, A_k.x - b_k)]^2$, where k indexes the constraints $Ax \geq b$. Then the relaxed problem is

$$\begin{aligned} &\min cx + p_\lambda(x) \\ \text{subject to}: \quad &\sum_{j=1}^{n_i} x_{ij} = 1, \quad i = 1, 2, \ldots, m \\ &x \geq 0, \text{ integer.} \end{aligned} \tag{PP_λ}$$

The key results are:

Result: (Weak Duality) For all $\lambda \geq 0$, $v(PP_\lambda) \leq v(MCIP)$.

Result: For a given $\lambda \geq 0$, if x is optimal to (PP_λ) and $Ax \geq b$, then x is optimal to $(MCIP)$. Further, for a given $\lambda \geq 0$, if x is ϵ–optimal to (PP_λ) and $Ax \geq b$, then x is ϵ–optimal to $(MCIP)$.

Result: (Strong Duality) If $(MCIP)$ is feasible, then there exists $\bar{\lambda}$ such that, for all $\lambda \geq \bar{\lambda}$, there exists $x \in S_\lambda$ such that $p_\lambda(x) = 0$ and x is optimal to $(MCIP)$, where S_λ is the set of optimal solutions to (PP_λ).

This result assures us that there exists a multiplier that returns an optimal solution to the original problem. Further, the proof guides us in finding that multiplier vector.

The relaxed problem, PP_λ, is solved heuristically by a new genetic algorithm designed to exploit the multiple-choice structure. Specifically, solutions of (PP_λ) are represented compactly in a manner that guarantees closure of the feasible region under the crossover operator. That is, if two parents are feasible, then the offspring is feasible.

This penalty function has appeared in the continuous nonlinear optimization literature. There, strong duality requires an assumption of convexity. In the integer case we get the strong result without convexity.

Traditional genetic algorithms use a roulette wheel approach to parent selection. Parents are chosen with likelihood proportional to their objective function value. Advantages are mimicry of biological genetics and an obvious tendency toward survival-of-the-fittest. The disadvantage is that the best solution in the population is not monotonically improving.

In the multiple-choice genetic algorithm we use an elitist strategy followed by random parent selection. The best few members of the population are automatically copied to the next generation. Parents are then chosen randomly from the entire old population. Advantages are that the best elements of the population are monotonically improving and that the random parent selection leads to certain conditional independencies when doing a probabilistic analysis of the algorithm. The disadvantage is the lack of biological justification. Both elitist strategies and random parent selection have appeared in the genetic algorithm literature (Goldberg [1989]).

Traditional 1-point and related crossovers are described above. We employ a Bernoulli crossover in which realizations from independent Bernoulli random variables choose which parent contributes the allele for each gene. This approach is called *parametrized uniform crossover* in Spears and DeJong [1991].

Traditional mutation involves a low probability random alteration of an allele. We employ a massive mutation operation that we call *immigration*. In this case we do not alter individual genes, but generate complete random individuals in each generation. Their genetic material is then interspersed in the genetic pool in future generations. Immigration serves the same purposes as mutation: replacing lost genetic material and resisting premature convergence. To its advantage, it is more easily analyzed probabilistically.

Integer programs are solved by a primal-dual process in which primal variables (x) are evolved with genetic algorithms and dual variables (λ) are evolved by a simplistic subgradient search. These searches are carried out harmoniously resulting in convergence to optimal primal and dual solutions. We have shown that:

Result: (Convergence) The genetic algorithm described above converges in probability to an optimal solution.

The algorithm was tested on problems designed to be very difficult. Specifically, they have many optimal solutions so that branch-and-bound algorithms get little guidance from bounds. To summarize the results, over 10 massively dual degenerate, 5000 variable multiple-choice integer programming problems (10 random seeds each) the GA averages 168 generations and 147 CPU

seconds on an IBM RISC/6000-730 to get within 5% of the linear relaxation bound. When the IBM package OSL is run as a heuristic, stopping with the same criteria as the GA, it averages 319 CPU seconds on the same machine. To prove optimality, OSL could not finish any problem within 20 hours. Hence, the GA makes a substantive contribution to our ability to solve large, complex integer programs.

In Hadj-Alouane, Bean and Murty [1993] we generalize to a specific nonconvex integer programming formulation of a task allocation problem arising in automotive design. Real problems of 480 variables were solved in an average of 310 seconds on an IBM RS/6000-320H.

The strong duality theory (Hadj-Alouane and Bean [1992], Bean and Hadj-Alouane [1992]) has substantial promise beyond the genetic algorithm used to date. The dualizing of the general constraints as described above is independent of the solution approach used. It might also be combined with traditional operations research approaches such as branch-and-bound or dynamic programming; and AI techniques such as simulated annealing and tabu search.

BIBLIOGRAPHY

Bean, J. [1994], "Genetics and Random Keys for Sequencing and Optimization," **ORSA Journal on Computing**, Vol. 6, pp. 154-160.

Bean, J. and A. Hadj-Alouane [1992]. "A Dual Genetic Algorithm for Bounded Integer Programs," Technical Report 92-53, Dept. of Ind. and Oper. Eng., University of Michigan, Ann Arbor, MI, 48109. To appear in **R.A.I.R.O.-R.O.**

Biegal, J. and J. Davern [1990], "Genetic Algorithms and Job Shop Scheduling," **Computers and Industrial Engineering**, Vol. 19, pp. 81-91.

Goldberg, D. E. [1989], **Genetic Algorithms in Search Optimization and Machine Learning**, Addison Wesley.

Goldberg, D. E. [1991], "Real-coded Genetic Algorithms, Virtual Alphabets, and Blocking," **Complex Systems**, Vol. 5. 139-167.

Hadj-Alouane, A. and J. Bean [1992], "A Genetic Algorithm for the Multiple-Choice Integer Program," Technical Report 92-50, Dept. of Ind. and Oper. Eng., University of Michigan, Ann Arbor, MI. 48109. To appear in **Operations Research.**

Hadj-Alouane, A., J. Bean and K. Murty [1993], "A Hybrid Genetic/Optimization Algorithm for a Task Allocation Problem," Technical Report 93-30, Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor, MI, 48109-2117.

Herrmann, J. and C. Lee [1993], Global Job Shop Scheduling for Semiconductor Test Operations," **ORSA/TIMS Phoenix**, November, 1993.

Holland, J. H. [1975], **Adaptation in Natural and Artificial Systems**, University of Michigan Press.

Kanet, J. J. and V. Sridharan [1991], "PROGENITOR: A Genetic Algorithm for Production Scheduling," **Wirtschafts Informatik**, Vol. 33, pp. 332-336.

Lee, C. and S. Piramuthu [1994], "Global Job Shop Scheduling with Genetic Algorithm and Machine Learning," **ORSA/TIMS Detroit**, October, 1994.

Michalewicz, Z. and C. Janikow [1994], "Handling Constraints in Genetic Algorithms,"**Proceedings of the Fourth International Conference on Genetic Algorithms**, pp. 151-157.

Norman, B. and J. Bean, [1994], "Random Keys Genetic Algorithm for Job Shop Scheduling," Technical Report 94-5, Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor, MI, 48109.

Spears, W. M. and K. A. De Jong [1991], "On the Virtues of Parameterized Uniform Crossover," **Proc. of the Fourth International Conference on Genetic Algorithms**, pp. 230-236.

Storer, R. H., S. D. Wu. and R. Vaccari [1992a], "New Search Spaces for Sequencing Problems With Application to Job Shop Scheduling," **Management Science**, Vol. 38, No. 10, pp. 1495-1509.

Storer, R. H., S. D. Wu, and I. Park [1992b], "Genetic Algorithms in Problem Space for Sequencing Problems," **Proceedings of a Joint US-German Conference on Operations Research in Production Planning and Control**, pp. 584-597.

Syswerda, G. [1991], "Schedule Optimization Using Genetic Algorithms," in **Handbook of Genetic Algorithms**, L. Davis (ed), Van Nostrand, pp. 332-349.

Joint Workshop on Artificial Intelligence and Operations Research
Timberline, Oregon, June 6-10, 1995

Interdependence of Methods and Representations in Design of Software for Combinatorial Optimization

Collette Coullard
Robert Fourer

Department of Industrial Engineering and Management Sciences
Northwestern University
Evanston, Illinois 60208-3119

Practical algorithmic *methods* for combinatorial optimization problems cannot be considered in isolation from the *representations* that people employ in communicating these problems to computer systems. Different representations define different problem classes, for which quite distinct types of methods are appropriate. Conversely, different methods have different ranges of applicability, which have motivated a diverse variety of representations.

This strong interdependence of method and representation in combinatorial optimization is quite the opposite of what one finds in linear or continuous nonlinear programming, where a single standard form permits communication between numerous methods and representations that have been independently developed. One consequence has been the tendency of separate research communities — such as "AI" and "OR" — to address optimization in separate ways, through different preferred combinations of representations and methods. Another consequence is a lack of good general purpose combinatorial optimization software, as one can see by examining the many ads for optimization systems in a typical issue of *OR/MS Today.*

In this presentation, we survey three representations popularly applied in combinatorial optimization — algebraic modeling languages, constraint logic programming languages, and network diagrams — and describe the kinds of optimization methods most commonly associated with these representations. Each pair of representations is then considered, to show how each one has been advantageous and how its advantages have begun to influence the design of the other. Our current research projects are described in conjunction with several of these comparisons.

Algebraic modeling languages. We take it for granted that virtually any problem of optimizing a linear function of given decision variables, subject to linear equations and inequalities in the variables, can be solved by one of just a few general-purpose algorithms. This remarkable property has made possible the development of comprehensive modeling languages for the support of linear programming. The user of such a language controls the structure of data and variables, which may be organized into any convenient combination of lists, tables, and more complex entities.

Our particular concern is with algebraic modeling languages, which have exhibited the greatest potential for extension beyond linear programming. Algebraic languages are based on familiar mathematical terminology for functions and comparisons, so that one may for example say:

```
minimize total_cost:
   sum {i in ORIG, j in DEST, p in PROD} cost[i,j,p] * Trans[i,j,p];

subject to Supply {i in ORIG, p in PROD}:
   sum {j in DEST} Trans[i,j,p] = supply[i,p];
```

By admitting nonlinear expressions and integrality restrictions on variables, the languages of AIMMS [3], AMPL [12, 13], GAMS [6, 7], and others can also describe problems of smooth nonlinear programming and of integer programming — two areas for which general-purpose algorithms have also become increasingly powerful. The AMPL language has been further extended to handle piecewise-linearities [11], while both AIMMS and AMPL allow network flow optimization problems to be formulated in terms of nodes and arcs [3, 11].

Constraint logic programming languages. Languages designed for the purpose of describing logical relationships can be extended in a natural way to encompass combinatorial optimization. Lauriere's ALICE [27], the most notable early work in this area, describes an optimization problem in terms of finding a best function of a certain kind; constraints are described through a variety of algebraic and logical forms. The work of Van Hentenryck and others in the context of the CHIP project [10, 36, 37], based on the Prolog language [35], has more recently attracted attention. Other examples are Colmerauer's Prolog III [8], and McAloon and Tretkoff's 2LP [29].

Problems expressed in constraint logic programming languages are typically solved by sophisticated general-purpose search methods, similar in nature to the methods that have long been employed in Prolog systems. Successful optimization requires that these methods be enhanced by a variety of backtracking, lookahead, and other strategies that can be controlled by the modeler or that can take advantage of problem-specific information that the modeler supplies.

Network-based representations. One of the largest and best-known classes of combinatorial optimization problems are those that can be posed in terms of networks: nodes, arcs connecting nodes, and data associated with arcs and nodes. Glover, Klingman and Phillips have formalized this representation, introducing the term *netform* to denote a general network data structure together with conventions for representing instances of the structure as network diagrams [15, 16]. Steiger, Sharda and Leclaire's GIN [33, 34] implements netforms within a model-management system for minimum-cost network flow problems; other examples include Ogryczak, Studziński and Zorychta's DINAS/EDINET [31, 32], McBride's NETSYS [30], Jones's NETWORKS [20, 21, 22, 23], and Kendrick's PTS [26].

The bulk of research in network optimization has proceeded by identifying a problem, devising an algorithm for that problem, and providing performance guarantees for the algorithm; the state of the art is well described for network flow problems by Ahuja, Magnanti, and Orlin [1], and for a variety of other network-based problems by Cook, Cunningham, Pulleyblank, and Schrijver [9]. This work has produced literally hundreds of good but narrowly targeted algorithms that have only very limited application, while modelers and analysts continue to be confronted with new network problems for which efficient and effective methods are not yet known. Certain heuristic methods do offer broader applicability, but they tend to

be of uncertain effectiveness; the most widely applicable approaches, such as genetic algorithms [28], simulated annealing [38] and tabu search [14], are unfortunately often very slow, and may produce no useful bounds on the quality of the solution returned.

Algebraic vs. constraint logic languages. For combinatorial optimization problems that are directly concerned with numerical decision variables, algebraic modeling languages offer a natural form of expression based on familiar mathematical notation, while logic programming languages often have the disadvantage of requiring a substantial re-thinking and translation. For other kinds of combinatorial optimization problems, however, the conversion to decision variables can be awkward, with the result that the logic programming approach has advantages both in naturalness of expression and in speed of solution.

The expressiveness of algebraic modeling languages might be extended in a number of ways by adding logical functions and operators and by extending the contexts in which they may be used. By merely allowing such operators as `or` and `if ... then ... else` to be applied more generally to variables, algebraic languages can naturally express a great variety of combinatorial constraints that would otherwise require the introduction of formulation tricks involving zero-one variables. The major obstacle to such extensions has been a lack of computational methods sufficiently general to address the great variety of combinatorial/algebraic problems that can result. Ideas from constraint logic programming methods may help to overcome this obstacle, and already there has been considerable interest in the relationship between various logic programming search strategies and established branch-and-bound strategies for integer programming [18, 19].

We have investigated another kind of extension to algebraic modeling languages, in which the optimization may be accomplished by selection of a subset or subsequence from a certain *decision set* (of cities, machines, bins, or whatever) in contrast to the assignment of values to decision variables [4]. Here again our approach has parallels to methods employed for constraint logic programming. We start with a general subset or subsequence enumeration method, and add a rich variety of directives by which the modeler can assist the search. Expressions within the directives can refer to problem-specific data and to aspects of the current state of the search.

The expressiveness of constraint logic programming languages might be increased in an analogous way, by adding some constructs better suited to the numerical-valued variables of algebraic modeling languages. Here the obstacle is likely to be of an opposite nature: the lack of connections between the logic programming systems and fast, robust solvers for such things as LP subproblems.

As languages of both kinds begin to overcome the obstacles we mention, they will inevitably gain more features in common. Eventually the distinction between them may not seem so great.

Algebraic vs. network-based representations. Although there are many network flow models that can be expressed as linear programs, the "flow in equals flow out" balance constraints at the nodes remain awkward to express in the customary terms of an algebraic modeling language. This was our motivation for including `node` and `arc` declarations to the AMPL language [11].

Many other important network problems give algebraic modeling languages even

more trouble. Their LP formulations may grow exponentially in the number of variables and constraints, while their integer programming formulations may involve constraints that have little obvious relation to the original problem. The simple minimum spanning tree problem, which admits very fast specialized algorithms, is a notorious example of this kind. To handle these cases in a natural way, algebraic languages will require more ambitious extensions. As an example, the hierarchical sets proposed by Bisschop and Kuip [5] could help with network problems, as they provide a natural way to deal with tree structures.

Algebraic and network representations are fundamentally distinct, however, and so are ultimately more likely to complement each other than to adopt each other's features. Jones and D'Souza [24] describe how a system based on manipulation of graphs for network flow modeling could be extended to interface with a system for algebraic modeling. MIMI/G, although its model representations are not algebraic, offers another idea of what might be possible; the modeler can define in great detail the appearance of a graphical network representation of a model's data and results; the network is "live" in that changing the graph on the screen will change the values in the tables. A more general possibility, explored by several investigators [2, 17, 25], is for the algebraic problem statement and the network diagram to become just two of many *views* of a model between which the user can switch as desired.

Logic-based vs. network-based systems. Although these systems address some of the same problems, they tend to take opposing approaches toward providing optimization methods to the modeler. Each might benefit by considering an approach that somewhat more like the other's.

Reflecting the general state of network optimization research cited previously, network-based systems have tended to rely on a *toolkit* of narrowly targeted methods. The user chooses from a menu of built-in models, constructs a network to be interpreted according to the chosen model, and then finally chooses from a predetermined list of methods that are applicable to solve the resulting problem instance. Such an arrangement has been highly successful for statistical packages, but has not been as effective in network optimization — perhaps because the range of models and methods is so much greater. Rather than continuing to expand their toolkits, designers of network-based optimization systems may benefit by considering some of the more general-purpose search methods employed in constraint logic programming; these methods may suggest approaches for putting network toolkits to work in a broader way.

Reflecting their origins in logic programming and artificial intelligence, logic-based optimization systems tend to adopt heuristic search approaches that are very widely applicable. By virtue of their generality, however, these search methods do not typically incorporate much specific knowledge about networks. They would not know, for example, that there are very fast methods for maximum flow or minimum spanning tree, which might be used in determining good bounds or feasible solutions as part of a search strategy. Just as network-based systems may need to become less problem-specific, logic-based systems may have to take a more problem-specific view to improve their effectiveness within an area such as network optimization. Some recent investigations pertaining to Horn clauses and related logical structures may prove useful in this regard.

References

[1] R.K. AHUJA, T.L. MAGNANTI and J.B. ORLIN, *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, Englewood Cliffs, NJ (1993).

[2] D. BALDWIN, The Development and Architecture of DOVE: A Multiview Viewpoint. *Proceedings of ISDSS Conference,* Austin, TX (1990).

[3] J.J. BISSCHOP and R. ENTRIKEN, *AIMMS: The Modeling System.* Paragon Decision Technology, Haarlem, The Netherlands (1993).

[4] J.J. BISSCHOP and R. FOURER, New Constructs for the Description of Combinatorial Optimization Problems in Algebraic Modeling Languages. Memorandum No. 901, Faculty of Applied Mathematics, University of Twente, The Netherlands (1990, revised 1994); forthcoming in *Computational Optimization and Applications.*

[5] J.J. BISSCHOP and C.A.C. KUIP, Hierarchical Sets in Mathematical Programming Modeling Languages. *Computational Optimization and Applications* **1**:4 (1992).

[6] J.J. BISSCHOP and A. MEERAUS, On the Development of a General Algebraic Modeling System in a Strategic Planning Environment. *Mathematical Programming Study* **20** (1982) 1–29.

[7] A. BROOKE, D. KENDRICK and A. MEERAUS, *GAMS: A User's Guide, Release 2.25.* Boyd & Fraser/The Scientific Press, Danvers, MA (1992).

[8] A. COLMERAUER, An Introduction to Prolog III. *Communications of the ACM* **33** (1990) 69–90.

[9] W. COOK, W. CUNNINGHAM, W. PULLEYBLANK and A. SCHRIJVER, *Combinatorial Optimization.* Unpublished manuscript (1994).

[10] M. DINCBAS, H. SIMONIS and P. VAN HENTENRYCK, Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming* **8** (1990) 75–93.

[11] R. FOURER and D.M. GAY, Expressing Special Structures in an Algebraic Modeling Language for Mathematical Programming. Technical Report 91-01, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL (1991); forthcoming in *ORSA Journal on Computing.*

[12] R. FOURER, D.M. GAY and B.W. KERNIGHAN, A Modeling Language for Mathematical Programming. *Management Science* **36** (1990) 519–554.

[13] R. FOURER, D.M. GAY and B.W. KERNIGHAN, *AMPL: A Modeling Language for Mathematical Programming.* Boyd & Fraser/The Scientific Press, Danvers. MA (1992).

[14] F. GLOVER, Tabu Search – Part II. *ORSA Journal on Computing* **2** (1990) 4–32.

[15] F. GLOVER, D. KLINGMAN and N.V. PHILLIPS, Netform Modeling and Applications. *Interfaces* **20**:4 (1990) 7–27.

[16] F. GLOVER, D. KLINGMAN and N.V. PHILLIPS, *Network Models in Optimization and Their Applications in Practice.* John Wiley & Sons, New York (1992).

[17] H.J. GREENBERG and F.H. MURPHY, Views of Mathematical Programming Models and their Instances. Technical Report, Mathematics Department, University of Colorado at Denver, CO (1992); forthcoming in *Decision Support Systems.*

[18] J.N. HOOKER, Logic-Based Methods for Optimization. *Operations Research Society of America Computer Science Technical Section Newsletter* **15**:2 (1994) 4–11.

[19] J.N. HOOKER, H. YAN, I.E. GROSSMANN and R. RAMAN, Logic Cuts for Processing Networks with Fixed Costs. *Computers and Operations Research* **21** (1994) 265–279.

[20] C.V. JONES, An Introduction to Graph-Based Modeling Systems, Part I: Overview. *ORSA Journal on Computing* **2** (1990) 136–151.

[21] C.V. JONES, An Introduction to Graph-Based Modeling Systems, Part II: Graph Grammars and the Implementation. *ORSA Journal on Computing* **3** (1991) 180–206.

[22] C.V. JONES, Attributed Graphs, Graph-Grammars and Structured Modeling. *Annals of Operations Research* **38** (1992) 281–324.

[23] C.V. JONES, An Integrated Modeling Environment Based on Attributed Graphs and Graph-Grammars. *Decision Support Systems* **10** (1993) 255–275.

[24] C.V. JONES and K. D'SOUZA, Graph-Grammars for Minimum Cost Network Flow Modeling. Technical Report, Faculty of Business Administration, Simon Fraser University, Burnaby, BC (1992).

[25] D.A. KENDRICK, Parallel Model Representations. *Expert Systems With Applications* **1** (1990) 383–389.

[26] D.A. KENDRICK, A Graphical Interface for Production and Transportation System Modeling: PTS. *Computer Science in Economics and Management* **4** (1991) 229–236.

[27] J.-L. LAURIERE, A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence* **10** (1978) 29–127.

[28] G.E. LIEPINS and M.R. HILLIARD, Genetic Algorithms: Foundations and Applications. *Annals of Operations Research* **21** (1989) 31–57.

[29] K. MCALOON and C. TRETKOFF, 2LP: Linear Programming and Logic Programming. In V. Saraswat and P. Van Hentenryck, eds., *Principles and Practice of Constraint Programming,* The MIT Press, Cambridge, MA (1995) 99–114.

[30] R.D. MCBRIDE, NETSYS — A Generalized Network Modeling System. Technical Report, University of Southern California, Los Angeles, CA (1988).

[31] W. OGRYCZAK, K. STUDZIŃSKI and K. ZORYCHTA, DINAS: A Computer-Assisted Analysis System for Multiobjective Transshipment Problems with Facility Location. *Computers and Operations Research* **19** (1992) 637–647.

[32] W. OGRYCZAK, K. STUDZIŃSKI and K. ZORYCHTA, EDINET — A Network Editor for Transshipment Problems with Facility Location. In O. Balci, R. Sharda and S.A. Zenios, eds., *Computer Science and Operations Research: New Developments in their Interfaces,* Pergamon Press, New York (1992) 197–212.

[33] D. STEIGER, R. SHARDA and B. LECLAIRE, Functional Description of a Graph-Based Interface for Network Modeling (GIN). In O. Balci, R. Sharda and S.A. Zenios, eds., *Computer Science and Operations Research: New Developments in their Interfaces,* Pergamon Press, New York (1992) 213–229.

[34] D. STEIGER, R. SHARDA and B. LECLAIRE, Graphical Interfaces for Network Modeling: A Model Management System Perspective. *ORSA Journal on Computing* **5** (1993) 275–291.

[35] L. STERLING and E. SHAPIRO, *The Art of Prolog: Advanced Programming Techniques,* 2nd ed. MIT Press, Cambridge, MA (1994).

[36] P. VAN HENTENRYCK, *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA (1989).

[37] P. VAN HENTENRYCK, A Logic Language for Combinatorial Optimization. *Annals of Operations Research* **21** (1989) 247–273.

[38] P.J.M. VAN LAARHOVEN and E.H.L. AARTS, *Simulated Annealing: Theory and Applications.* D. Reidel, Norwell, MA (1987).